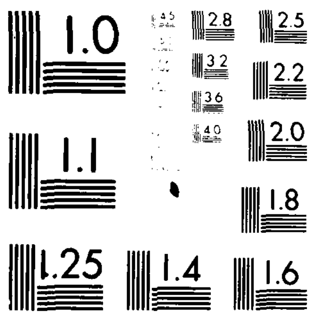
AD-A113 172 BOLT BERANEK AND NEWMAN INC CAMBRIDGE MA F/G 9/2

DISTRIBUTED OPERATING SYSTEM DESIGN STUDY. VOLUME I.(U)

JAN 82 H C FORSDICK, W I MACGREGOR F30602-79-C-0193

UNCLASSIFIED BBN-4674-VOL-1 RADC-TR-81-384-VOL-1 NL

1 OF 3


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

(12)

AD A113172

RADC-TR-81-384, Vol I (of two)
Final Technical Report
January 1982

DISTRIBUTED OPERATING SYSTEM DESIGN STUDY

Bolt Beranek & Newman, Inc.

Harry C. Forsdick
William I. MacGregor
Richard E. Schantz
Steven A. Swernofsky
Robert H. Thomas
Stephen G. Toner

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

82 04 09 055

RADC-TR-81-384, Vol I (of two) has been reviewed and is approved for publication.

APPROVED: [illegible]
THOMAS F. LAWRENCE
Project Engineer

APPROVED: [illegible]
JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER: [illegible]
JOHN P. HUSS
Acting Chief, Plans Office

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-81-384, Vol I (of two)	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) DISTRIBUTED OPERATING SYSTEM DESIGN STUDY		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report 12 Jul 79 - 1 May 81
		6. PERFORMING ORG. REPORT NUMBER 4674
7. AUTHOR(s) Harry C. Forsdick Steven A. Swernofsky William I. MacGregor Robert H. Thomas Richard E. Schantz Stephen G. Toner		8. CONTRACT OR GRANT NUMBER(s) F30602-79-C-0193
9. PERFORMING ORGANIZATION NAME AND ADDRESS Bolt Beranek and Newman, Inc. 50 Moulton Street Cambridge MA 02138		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F 55812110
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COTD) Griffiss AFB NY 13441		12. REPORT DATE January 1982
		13. NUMBER OF PAGES 278
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Thomas F. Lawrence (COTD)

19. KEY WORDS (Continue on reverse side if necessary and identify by bloc

Distributed Operating System
Network Operating System Resource Control
System Reliability

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Three specific issues regarding the design of an operating system to control information processing resources distributed throughout a network of computers were investigated. These issues are: (1) techniques which can be used to maintain processing continuity of the distributed system (2) the policies and mechanisms to be used for resource allocation among system processes and (3) characteristics which constituent computers should possess to be cooperative elements with a distributed system.

(over)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Resource allocation policies are discussed and simulations of several resource allocation schemes were completed.
In addition, individual techniques to provide for processing continuity are described along with a description of how these techniques could be applied to system operation. Lastly, recommendations for future relating to the above issues is provided.

TABLE OF CONTENTS

1. EXECUTIVE SUMMARY

Computer communication and computer architecture technology has advanced to the state where it is feasible to build computer systems from many smaller systems. Such distributed architectures are attractive for many reasons including:

- _Economics_. The effort spent on operational software must not be lost. Often an important function of a new distributed system is to integrate information gathered or processed by several existing systems.
- _Reliablity_. Distributed systems built from autonomous components can be made more reliable than single component systems.
- _Scalability_. Building a system out of repeatable parts makes it possible to adjust the size of a system without reprogramming all of the parts.
- _Inherent Separation_. Some applications must deal with parts that are not physically close together. There is often no alternative to building these systems from separate parts.

Achieving the potential benefits of a distributed architecture involves more than interconnecting separate computer systems. The allocation and sharing of resources among the multiple components of a distributed system must be managed. This is the purpose of a Distributed Operating System (DOS).

The Distributed Operating System Design Study was aimed at investigating issues encountered in the development of such operating systems. The goal of the project was to investigate three important distributed system design areas:

1. Failure detection and recovery.

   The redundancy, parallelism, independent failure patterns, and broadcast capabilities exhibited by distributed architectures provide a potential for highly reliable systems. However, current distributed systems often display worse or only marginally better system reliability than their single host counterparts. The natural approach of patterning distributed systems after more familiar single host systems often leads to the "in series" connection of system components which results in vulnerability to single component failures. Furthermore, the reliability mechanisms described in the literature for distributed environments typically either are designed to solve a very specific problem as opposed to a general systemic one, or are difficult to integrate with other system mechanisms. One objective of the DOS Design Study was to investigate solutions to reliability problems which exploit the potential for reliability inherent in distributed architectures and which also can be integrated with other system mechanisms.

2. Global system control and scheduling.

   Current distributed systems frequently exhibit noticeably worse performance characteristics than their single host counterparts. This is due, in part, to a lack of global resource management and system control in these systems. For example, when components on two hosts must exchange information, the components should be scheduled to receive processor resources at (or about) the same time. The processor resource is just one of many resources that could be allocated according to global needs. A second objective of the project was to investigate the problem of coordinated resource management in distributed systems.

3. Features desirable for DOS constituent hosts.

   Distributed systems are ultimately implemented from the features and capabilities provided by the underlying constituent hosts. A problem with many single host operating systems is that they do not allow their resources to be used in ways other than those built into the operating system. A third objective of the DOS Design Study was to characterize the capabilities host operating systems should provide to facilitate their integration into a DOS.

Early in the study we discovered that we needed examples of the demands applications place on various reliability and resource management mechanisms in order to make realistic decisions about their effectiveness or applicability. We developed descriptions of three hypothetical applications, suitable for implementation on a distributed hardware base, which served as a framework for much of the study. The applications were chosen to reflect a broad range of design constraints with respect to system performance, cost, reliability, and modularity.

The example applications were:

1. Field Command Unit. This hypothetical application was chosen to be representative of a tactical command and control environment. It is a system intended to support the mission of a unit staffed by command officers and teams of specialists in intelligence, logistics, planning, air support, etc. Modularity, survivability and mobility are important system requirements.

2. Air Traffic Control. This application focuses on supporting air traffic controllers responsible for controlling aircraft traffic over a large area. The goals of the system are the protection of aircraft occupants, timely arrival of flights, and fuel conservation.

3. A Military Message System. This example considers an electronic message system for a military command center. The purpose of the system is to help the command staff compose, send, read and file messages in

a way that supports the command's normal routine. Both intra- and inter- command center message traffic is to be supported.

We believe that these example applications serve to motivate a broad range of requirements for military and commercial systems. However, the "designs" developed for these examples should be regarded as purely hypothetical.

One of the first tasks undertaken was to survey reliability mechanisms for distributed and centralized systems. Over twenty techniques were studied and analyzed.

These mechanisms are generally designed to satisfy one of two reliability goals (although some are designed to satisfy both):

1. To ensure _correct operation_ in the presence of component failures.
2. To ensure _continuity of operation_ and task completion in the presence of component failures.

Many of the mechanisms studied were originally developed for centralized systems and reformulated for use in a distributed environment. This could be done because many reliability issues are important in both environments. There are, however, three important ways in which reliability considerations for distributed environments differ from those for centralized environments:

1. The ability to recover from failures is a more critical concern for a distributed environment. A simple probabilistic argument can be used to show that in order for a distributed architecture to provide the same level of "system" reliability as a centralized one its design must include mechanisms which permit it to function as a system when some of its components are non-functional.

2. A distributed environment can be engineered so that component failures are independent. A consequence of independent failure patterns is that it is possible, at least in principle, to engineer systems that are highly fault tolerant since the components which remain operational can be used to provide continuity in system service.

3. Failure recovery in a distributed environment requires coordination between the components. As a result failure recovery techniques used in distributed systems require additional mechanism to accomplish the coordination.

The types of failures or errors that can be expected in a distributed environment can be divided into three categories:

- o Host outages.
- o Communication outages.
- o Component malfunctions.

This study focused on recovery from host and communication outages. Although recent work on the important problem of recovering from component malfunctions appears promising, more work is required to refine and extend the emerging results for integration into system designs.

Analysis of the various reliability techniques resulted in the identification of a relatively small number of key ideas that reappear in slightly different ways in different techniques. The key ideas include:

- o Redundancy of data and control.
- o Atomicity.
- o Isolation of partial results.
- o Guaranteed permanance of effect.
- o Restoration of an acceptable state.
- o Bounding the time during which a failure causes problems.
- o Use of timeouts

In general terms, techniques concerned with "correct" operation tend to focus on being "careful" about how operations are performed, whereas techniques that address continuity of operation focus on using alternate processing or data sources to proceed, perhaps with limited functionality, toward task completion.

As a result of this survey we concluded that existing approaches to reliability appear to be adequate as building blocks for reliable distributed systems. However, most of the mechanisms surveyed were formulated apart from consideration of how they might fit in with other reliability mechanisms or other operating system features.

We next focused on this problem of integrating reliability mechanisms into distributed operating systems by developing a design for an integrated reliability system for a DOS. Our objective was a design capable of providing reliability for a DOS as a whole. This would require the engineering and careful integration of a number of different reliability techniques at different system levels. By doing this we hoped to learn how difficult it would be to integrate various reliabiity techniques with one another and with mechanisms which implement other important parts of the system.

Before we could proceed with the reliability system design it was necessary to develop a DOS architecture. Although our primary objective was to specify a DOS in sufficient detail to provide a framework for development of the reliability system, we were also interested in investigating how various DOS features interact with one another and with the reliability system.

The environment assumed for the DOS is a collection of computers interconnected by a high bandwidth, low delay local area network. The component host computers are assumed to be of different manufacture and to include:

- o <u>Shared general purpose hosts</u>, such as time sharing systems;
- o <u>Hosts dedicated to specific functions</u>, such as data storage or i/o device control, and possibly to specific applications such as signal processing;
- o <u>Terminal access hosts</u>, each of which provides a number of users with terminal access to the DOS;
- o <u>User work station hosts</u> which are each dedicated to a single user and provide both local computing and access to the other DOS resources.
- o A <u>gateway host</u> which functions to link the DOS cluster to other networks and clusters.

The DOS acts to coordinate the operation of the collection of hardware for the local network cluster.

We specified the DOS software architecture from three different but related viewpoints:

- The _programming interface_, which is concerned with the features provided to application programs by the DOS;
- The _user interface_, which is concerned with the features provided directly to human users of the DOS;
- The _implementation_, which is concerned with the major software modules and the interactions among them that are required to implement the DOS features.

In addition, we described how the DOS software architecture operates to support several common user scenarios.

The purpose of the DOS reliability system is to ensure reliable operation of the system and to provide mechanisms at the programming interface which can be used to construct reliable application software. Failure recovery is an important function of the reliability system. We found it useful to divide recovery into three parts:

1. _Failure detection_.
2. _Reconstitution_, whereby recovery is initiated by organizing the remaining operational resources into a functional system.
3. _Reconciliation_, which occurs after failed components are restored and integrated back into the system, and involves reconciling with the failed components the

results of any operations effecting the failed components that were performed while they were inaccessible.

The responsiblity of the recovery system is to ensure that the DOS operates in a way that makes reconstitution possible, and that recovery actions occur in a way that makes reconciliation feasible.

The basis of the DOS recovery system is a design principle and a carefully integrated set of reliability techniques. The _design principle_ is:

> A side effect of the normal operation of the system should be the generation and placement of information that makes recovery oprations possible.

The reliability techniques are:

- o Redundant processing capability
- o Redundant sources of critical data
- o Self-identifying data
- o Selective checkpointing
- o Hierarchies of recovery agents

In addition to developing a design for the DOS reliability system we illustrated by means of a number of examples how the reliability techniques in the design work together to ensure system reliabilty, and how they can be used to develop reliable applications.

When we surveyed the area of global resource management we discovered that, unlike reliability, very little work had been done in the area. During this study we focused on distributed resource management from three perspectives.

1. Formal specification of resource management policies.

   Without a clear and unambiguous statement of goals it is difficult to discuss the suitability of various global resource management mechanisms. We believe that most resource management policies can be conveniently described in terms of two elements: _priority_ and _relative service_. For this part of the effort we developed formal definitions for four different resource management policies: non-preemptive priority, preemptive priority, non-preemptive relative service, and preemptive relative service. The policies were formulated in terms of criteria for sequences of resource request and resource grant events. They were formulated first for central site systems and then extended for distributed systems. The effects of distribution on the formal policy definitions were:

   a. A time ordering of events at different sites is not readily available in a distributed system, but must be deduced from observations.

   b. Message transmission may be long, making it unrealistic to model message transmission by a single event.

   These factors contribute uncertainty to the time ordering of resource allocation events, complicating tests for policy compliance.

2. Specification of distributed resource management algorithms.

   Implementation of a distributed resource management scheme requires an algorithmic description of its operation. Several hosts may need to cooperate to ensure proper coordination of information about the state of the resource, its users, and its requests. We investigated the structure of distributed resource management algorithms by detailing four example algorithms that implement a non-preemptive priority management policy. We demonstrated that each algorithm, in fact, implements the non-preemptive priority policy, and compared and contrasted them with respect to a taxonomy of the design space.

3. Performance benefits of load sharing and priority resource management in distributed systems.

   Load sharing is often suggested as a means for improving the performance or increasing the capacity of a distributed system. However, before investing the effort to implement load sharing, convincing evidence should be found to justify the effort. One objective was to quantitatively study the benefits of load sharing. Another objective was to study the effect of preemptive priority resource management on response times and resource utilization. Our approach was to compare queueing system models of distributed systems, both with and without global assignment of tasks to processors, over a range of model parameters. For the simplest cases we were able to obtain analytic results. For more complex cases we used discrete event simulation, first validating the simulator on the cases for which we had obtained analytic results. The major direct findings of this study are that (for the models and work loads used) load sharing is always beneficial, but in many cases only modestly so, and that preemptive priority management in a distributed system is very effective in providing preferential treatment to certain users.

In each case we proceeded from general issues to specific examples, and tried to present a framework that will be useful for further research.

Experience has shown that some operating systems are much easier to integrate into a DOS than others. Part of this study attempted to identify operating system features that make it relatively easy for a host to function as a constituent host in a

DOS. The following is a list of system characteristics or features which facilitate integration of a host into a DOS. The features on the list are not strictly independent of one another. They are listed roughly in order of decreasing importance.

1. The network should be integrated into the system as a "standard" device, and all communication functions supported by the network should be accessible to application programs.

2. The system should be extensible. It should be possible to modify and augment the standard "system" functions by application level software in a way that is efficient and is transparent to users of the "extended" system functions.

3. The system should implement the notion of process in a way that permits application level software to create and manipulate processes. Processes should be relatively inexpensive to use.

4. The system should provide an efficient mechanism for communication among processes that reside within the same user "jcb" or that reside in different user "jobs".

5. The system should support "device independent" i/o in the sense that a process should be able to do certain "generic" i/o operations (e.g., read, write) with a device or with another process as a source or destination without detailed knowledge of its characteristics. Of course, device dependent i/o operations should also be supported.

6. The system should provide means for a process to reliably determine when data output to non-volatile (e.g., disk) storage has actually been successfully written onto the storage medium. This feature is important as the basis for a number of reliability mechanisms.

7. The system should provide means for (properly authorized) processes to control system resource management decisions. For example, a process

designated by a user should be able to assign (relative) priorities for other processes belonging to the user.

There are two system design philosophies which are perhaps as important as these specific features. The first is to recognize that system resources may need to be used in ways other than those anticipated by the system designer, and to design the system to facilitate or at least not prevent such use. The second is that the system should acknowledge that activities can occur outside of its direct control. For example, users may be authenticated by other hosts, and the host system should (with proper precautions) be prepared to accept such authentications.

As a result of this study we conclude the following:

- o Reliability mechanisms appropriate for use in distributed systems have been studied thoroughly in isolation and apart from implementations. In addition, this study developed a design for an integrated reliability system for a DOS. We believe that the next step should be to develop prototype implementations in order to experimentally verify the effectiveness and measure the performance and overhead of integrated reliability systems, such as the one formulated in this study.
- o Resource management and scheduling for distributed systems is a relatively unexplored area. This study made progress in the area but more research is needed before a unified treatment of resource management for distributed systems can be developed.
- o Operating system features that make it relatively easy to integrate a host into a DOS are reasonably well understood

In addition, we make the following recommendations:

- The extent to which system functionality may need to be limited during failure recovery in order to make reconciliation possible should be investigated.

  It is clear that one type of system degradation during failure mode operation results from limitations placed on certain aspects of system functionality. For example, although it may be possible to allow a user to create files when the host responsible for maintaining his file directory is inaccessible, the user cannot be given complete freedom in naming new files if reconciliation is to be possible when the directory host is again accessible. What is less clear are the ways in which and the extent to which system functions need to be limited in order to permit reconciliation.

- An effort tc integrate support for reliability mechanisms into programming languages should be undertaken.

  Apart from exception handling, current programming languages provide little support for reliability mechanisms. Language support for checkpointing is mentioned below. In addition, a language might support the notion of redundant or alternate sources of processing and data by facilitating the retrying (i.e., re-execution) of a "block" of code, possibly with slightly different parameters. A possible approach here would be to explore how support for reliability mechanisms might be integrated into a modern language such as Ada. Part of this effort should focus on developing a better understanding of the tradeoffs between additions of reliability features to a language (and its runtime support system) and application code implementations of reliablity mechanisms.

- Means should be developed for declaring the extent of a process state that needs to be saved as a checkpoint state for recovery purposes.

  The cost of the checkpoint/restart recovery mechanism is related to the size of the process state that must be saved. In many situations it is not necessary to save an entire process state (i.e., the entire process address space, internal registers, etc.) in order to be able to restart it. Furthermore, the parts of the state that need to be checkpointed may change as the process executes. Means should be provided, perhaps through a declarative statement in the programming language used to implement a process, to declare the extent of the checkpointable state.

- Programming language support for distributed application programs should be investigated.

  With current programming languages, the division of an application program into multiple parts for distribution, and the location and interconnection of distributed components must be expressed in an *ad hoc* manner outside of the programming language. As a result, compiler checking and optimization techniques are not applicable to the distributed components, the modularization of the multiple components is fixed and difficult to alter, and similar approaches to dealing with distributed components are re-invented for each application program. Programming language features for expressing the modularization, interconnection and interaction between distributed components of an application program would represent a significant advance in the state of the art.

- Mechanisms for authentication, access control, protection, and privacy for distributed systems should be investigated.

  Relatively little work has been done in the area of access control and protection for distributed environments. As we developed the DOS architecture used for this study, we had difficulty formulating an approach to access control. Authentication of an entity making a request for access to data or a service is central to access control. Reliable authentication of a remote entity is difficult in a distributed environment. Public key encryption has been suggested by some as a good basis for access control and protection in distributed systems. However, more work is required to develop an approach based on it. Furthermore, given the current state of the art, encryption and decryption using public keys are expensive operations, and may be impractical for operational use in access control and protection mechanisms. We believe this is a very important area where further research is needed.

- The development and formulation of resource management policies should be further investigated.

  This study developed a formalism for specifying resource management policies and then used it to define four different policies. There are many other policies which may be of interest. Further work is required to identify important policies and to represent them in a formalism such as developed in this study.

- An effort to develop user models of resource management policies should be undertaken.

  Computer users see a system from perspectives related to their tasks. A user should not have to understand low level system mechanisms to know what to expect of a given resource management policy. While this is a clearly desirable goal, it is typically not achieved by contemporary systems.

- More comprehensive performance analysis of resource management mechanisms for distributed systems should be undertaken.

  Simulation and/or analytic models can be used to evaluate the performance properties of resource management mechanisms. We believe that the work in this study represents a good basis for further research in this area.

The rest of this report presents the results of the DOS Design Study in detail. Appendix A is the first interim technical report for the study, and Appendix B is the second interim technical report.

APPENDIX A
INTERIM TECHNICAL REPORT 1

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Purpose of Phase I of the Study

Distributed systems are now a reality: it is rare that a single computer system is acquired without any plans for connecting it to some other computer system or computer communications network. Technology has advanced to the point where it is currently feasible to build computer systems out of many smaller computer systems. Such an approach is attractive for many reasons, some of which are:

- **Economics.**

  The effort expended in building current operational systems must not be lost. Quite often the function of a new distributed system is to integrate information gathered or processed by several existing systems.

- **Reliability.**

  Distributed systems built out of autonomous components provide alternatives in the event of a failure. By eliminating single critical failure points, distributed systems can be made more reliable than single component systems.

- **Scalability.**

  Building a system out of repeatable parts makes it possible to adjust the size of a system without reprogramming all of the parts.

- **Inherent Separation.**

  Some applications must deal with parts that are not physically close together. Many times there is no alternative to building these systems out of separate parts.

Distributed systems that realize these benefits have not, in general, been realized for two basic reasons. First, the coordination required to make multi-component systems operate in the manner described above has not been developed in a general, integrated fashion. For example, there is little coordination in the use of resources across the component systems of a distributed system. Second, some of the approaches to solving problems of an application that are feasible with distributed systems have not been feasible before. The necessary concepts and mechanisms for taking full advantage of the capabilities of distributed systems need to be developed. For example, autonomous operation of replicated parts of an application has only recently been possible through significant advances in computer interconnection technolcgy. The models and mechanisms developed for single host applications do not address replication of parts nor how to use replicated parts in a general way.

Thus, usable distributed systems are more than connections between separate computer systems. For successful sharing of resources to be achieved among the users of a distributed computer system, there must be programs (or parts of the operating system) whose purpose is to manage the allocation and sharing of resources among the multiple components of the distributed system. This is the purpose of a Distributed Operating System (DOS).

The Distributed Operating System Design Study is aimed at investigating issues encountered in the development of such operating systems. This report describes the results of Phase I of the Study in which requirements of characteristic applications and existing techniques for providing both reliable and coordinated access to distributed resources have been examined. In Phase II we will investigate some of the unsolved issues discovered during Phase I.

1.2 Emphasis on Reliability, Resource Control and Constituent Host Features

Two specific problems associated with existing distributed systems are the lack of global management of resources and the absence of integrated reliability and error recovery mechanisms. The issue of global scheduling first came to our attention in the context of the National Software Works (NSW) [8], a distributed operating system. We observed that one of the reasons the performance of NSW suffered was that there was no coordination among the NSW constituent hosts concerning the allocation of processor resources to NSW components. For example, if the Works Manager and the Foreman components were going to engage in an exchange of information, then the scheduling of these two processes should be coordinated so that they would receive processor resources at (or about) the same time. This example illustrates a general problem that distributed system designers must address: the uncoordinated management of component system resources. Processor resources are just one type of resource that needs to be allocated according to the global needs of a distributed application. One objective of Phase I of the DOS Design Study has been to investigate solutions to the distributed resource allocation problem that provide ways for component hosts to exchange information about the resource requirements of distributed computation. These solutions can also serve as the basis for more global control of distributed resources.

While distributed systems can provide increased functionality over their single host counterparts, they frequently display only marginal improvement in system reliability and noticeably worse performance characteristics. A natural approach to distributed system design is to pattern these systems after more familiar single host systems. This has often led to "in series" or tree structured connection of components which frequently results in single component failure points and performance bottlenecks. Reliability mechanisms that have been developed for distributed systems are usually designed to solve a specific problem without general purpose functionality, or are difficult to integrate with other system mechanisms. Improvements in the capability to recover from errors in distributed systems will require new models of usable programming primitives. These models will exploit the natural strengths of distributed systems such as redundancy, parallelism, broadcast capabilities, and relative ease of reconfiguration. A second objective of the DOS Design Study has been to investigate solutions to problems of reliability which not only make use of such natural strengths but which also can be integrated with other distributed system mechanisms.

Distributed systems must ultimately be implemented out of the features and capabilities provided by the constituent hosts of the system. A problem associated with many underlying host operating systems is that they do not allow the resources they manage to be used in any way other than that built into the operating system. For example, most single host operating systems provide file resources for use by application programs. These files are organized by a naming scheme that has a well defined syntax. Few operating systems allow for any alternative file naming syntax as might be required to support a distributed operating system that is built from multiple heterogeneous constituent hosts. A third objective of the DOS Design Study has been to characterize the set of capabilities that need to be supplied by the operating systems of the constituent hosts that are part of a distributed operating system.

1.3 Structure of Report

The results of Phase I of the study are presented in detail in the remainder of this report. Figure 1 illustrates the relationships between the chapters of this report. Chapter 2 describes the assumptions about the components, their interconnection and probable modes of failure. In Chapter 3, we describe three characteristic applications that could be implemented on top of a DOS. This description is intended to illustrate the diverse requirements of realistic application programs. Chapter 4 contains the results of our investigation of reliability mechanisms and their application to distributed systems. In Chapter 5 we consider global resource allocation requirements and possible strategies. The underlying features of a Constituent Operating System (COS) are the subject of Chapter 6. Several problems associated with reliability mechanisms and global resource control that are still outstanding serve as the basis for our continuing work in Phase II of the DOS Study. A

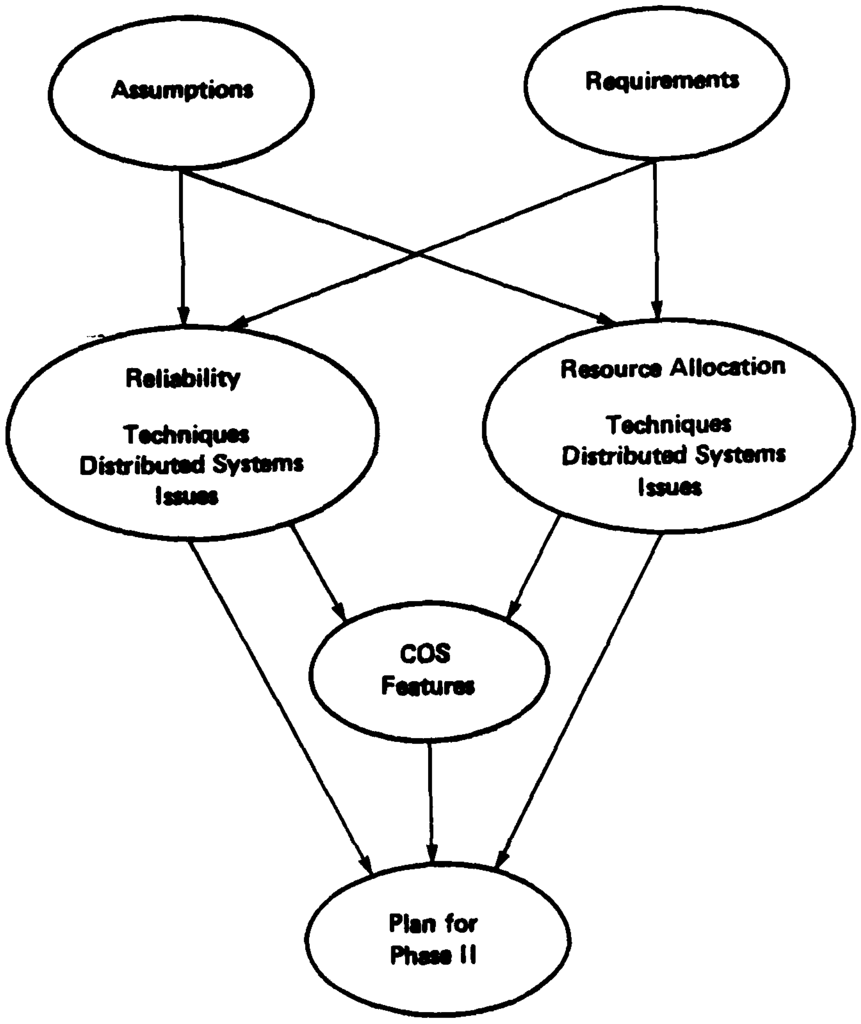


Figure 1. Relationships between Chapters

plan for Phase II is presented in Chapter 7.

2. ASSUMPTIONS

2.1 Introduction

The purpose of this chapter is to establish a context for the DOS Design Study by specifying an environment in which DOSs of interest must function, and by identifying the characteristics of the applications they should support.

2.2 Environment

The environment consists of clusters of host computers. Hosts within a cluster are separated by distances of up to a few kilometers, and are interconnected by means of a high speed local network (see Figure 2). The clusters are distributed over distances of tens to thousands of miles, and may communicate with one another by means of a geographically distributed network.

The details of the networks are not important to this project. However, some of the network characteristics are.

The intra-cluster network is assumed to operate in the 1-5 MB/sec range. Thus, while communication at high speed and with low delay between hosts can occur, communication within a host is several times faster. The Ethernet developed at Xerox Palo Alto Research Center [37] and the ring network developed at the

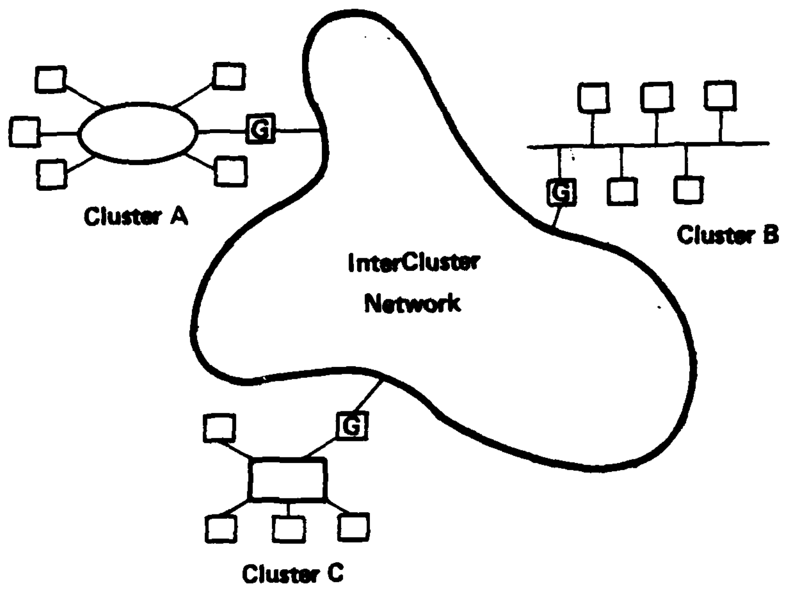


Figure 2. Interconnection of Clusters

University of California at Irvine [14] are examples of local networks of this type.

The network supporting communication between clusters is assumed to operate at lower data rates. Inter-cluster data rates in the 10-50 KB/sec range and inter-cluster delays in the 100 msec to 1 sec range are assumed. The ARPA network [52] is a good example of such a network.

The way these network characteristics are accomplished is not important to this study. Inter-cluster communication might, in fact, be supported by more than one network. Similarly, intra-cluster communication might be accomplished using several different communication media. However, for simplicity we shall speak of a single network within a cluster, and a single inter-cluster network.

We assume a standard addressing scheme for hosts which permits a host to address any other host in a uniform fashion regardless of whether the host addressed is in another or the same cluster.

The focus of this project is the activity that occurs inside a cluster. We assume that the clusters exhibit a great deal of autonomy with respect to one another at the system level.

Inter-cluster communication is assumed to occur infrequently relative to intra-cluster communication. When inter-cluster interactions do occur they usually fall into one of two major categories: access of data stored in another cluster or person-to-person messages between users of different clusters. Another way to say this is that the DOSs of interest for this study are operating systems for a cluster.

The host computers within a cluster are assumed to be of different manufacture and range from single-user to shared machines. Each host has its own operating system, assumed to be different from host type to host type. The operating systems are assumed to be state-of-the-art systems which have been augmented to operate in a network environment in the sense that each provides facilities for communicating with other hosts. However, any higher level inter-host functions are assumed to be functions provided by a DOS.

The size of a cluster may vary in terms of the number of users it must support, the number of functions it must provide, and, possibly, the amount of hardware redundancy it incorporates to achieve system reliability goals. Thus, the DOS for a cluster must permit modular expandability in the sense that it should allow clusters to be configured with a variable number of hosts.

2.3 Application Characteristics

Most activity within a cluster is assumed to be initiated by users at interactive terminals, and most of the applications are assumed to be interactive in nature.

Users within a cluster are assumed to be working toward a common set of goals. However, at the system level it is necessary to think in terms of multiple independent applications. Some activities may be deemed more important than other activities. Thus, tasks may have different priorities. Furthermore, task priorities may change dynamically.

Timely completion will be important for certain activities. By this we mean that some activities will have associated with them a time by which their results are needed. If the results are not produced by that time, the value of the results, and thus of the activity itself, is likely to decrease significantly. Note that by "timely completion" we do not mean real time processing.

Much of the activity within a cluster can be characterized as office automation. By this we mean that the cluster resources are used to maintain online, and to process, information required to support the mission of the cluster user community. More specifically, much of the cluster activity is document preparation, filing, simple computations, checking and modifying information in databases, form processing, intra-cluster person-to-person communication, and inter-cluster person-to-person communication.

We distinguish _office automation_ applications from more traditional _data processing_ applications in the following way.[1] A data processing system is used to implement a single locus of control which is a single component, not a collection of autonomous parts; the algorithm ordinarily proceeds without the need for human intervention. Typical data processing systems compute payrolls, implement accounting systems, or manage inventories. The office automation applications are made up of collections of highly interactive autonomous tasks that execute in parallel or at least asynchronously; these tasks include document preparation, document management, communication and aids for decision making.

Electronic mail will be an important application. Within a

[1]The remainder of this paragraph paraphrases the distinction between data processing and office automation suggested by Ellis and Nutt [11].

cluster it will be used as the basis for coordinating user activity. It will also serve as the means for interactions between users in different clusters. Many cluster tasks will be database intensive; various databases will be queried and updated as tasks proceed. Some applications will involve situation displays, where the information displayed is generated by the results of database queries and the execution of application programs.

In addition to office automation applications, the cluster can be expected to support some traditional business and scientific data processing applications.

We assume that the ability to support real time processing is not a DOS requirement. More specifically, we assume that the cluster DOS will not need to meet real time processing requirements, but that it, or applications it supports, may deal with the results of real time processing. That is, processors that provide real time processing services must be able to be integrated into the cluster DOS, but their real time behavior must be ensured by their host operating systems and not by the DOS. Chapter 3 discusses application characteristics in more detail and describes three example applications.

2.4 Failure Modes

System failure recovery and reliability mechanisms are, of course, designed to deal with a set of expected failures. The failures of interest to this project fall into two areas: host outages and communication outages. Mechanisms for recovering from program errors are not of interest in this project, except to the extent that such errors are indistinguishable from other failures.

Individual hosts within a cluster may fail. We assume that the hosts are sufficiently isolated by hardware and operating system software to ensure that the failure of one host will not directly cause the failure of another. Hosts generally are configured with two types of memory. For _volatile_ memory (e.g., semiconductor memory), the contents are lost after a host outage. For _non-volatile_ memory (e.g., disk), the contents usually remain intact after an outage. Occasionally the contents of non-volatile memory will be lost as the result of a host crash. In such cases, the memory will be restored to a previous state. We assume that DOS software will be able to detect when such a backup occurs. Hosts may remain unavailable for extended periods for maintenance or offline fault isolation. We assume a host mechanism which can accomplish "clean" shutdown.

Communication within a cluster may fail, causing some hosts to be unable to communicate with other cluster hosts. The network may occasionally fail to deliver a message to a host; for such intermittent failures the next message is likely to be successfully delivered. Longer term communication outages, characterized by periods during which no communication between sets of hosts may occur, can also be expected. Outages of inter-cluster communication can also be expected.

Host and communication outages can be characterized as permanent or temporary in terms of their duration relative to various DOS transactions. This characterization is related to the timeliness criteria for transactions. A permanent outage is one which persists for the duration of a transaction. That is, the failed component will not be re-integrated into the DOS before the transaction must be completed. Some transactions, because of their timeliness criteria, will not be able to wait for the restoration of failed communication or host services. This suggests that partial results or alternative sources of services or data should be used to complete these transactions. Other outages will be temporary in duration with respect to transactions in that the failed components will be restored to the system before the transaction must be completed.

The DOS should incorporate means to accomplish two somewhat different types of failure recovery. One would work to ensure transaction completion when failed resources required for a transaction become available. The other would have the goal of ensuring partial completion of transactions in a timely fashion by using partial results and alternative sources of data and processing. In practice, choosing the appropriate type of recovery for a transaction will involve an assessment of the expected duration of the outage(s) relative to the nature of the transaction and its timeliness criteria. This may be difficult, and for some transactions it might make sense to initiate both types of recovery. Chapter 4 discusses failure modes in more detail as well as generic and specific approaches for dealing with failures.

2.5 Other DOS Issues

There are other important DOS design issues in addition to those that are the subject of this project. These include:

- Communication security for intra-cluster and inter-cluster communication.
- Multi-level security within hosts and across hosts within a cluster.
- Access control within hosts.
- User and process authentication.

This study will not directly address these issues. We shall assume mechanisms exist or will be developed that address them adequately, and that are compatible with the approaches to reliability and resource allocation which are the focus of this project.

3. REQUIREMENTS OF CHARACTERISTIC APPLICATIONS

3.1 Introduction

Early in the study we discovered that in order to make realistic decisions about the effectiveness or applicability of various mechanisms, we needed some examples of the demands placed on these mechanisms and constraints on the resources available to a mechanism. As a result, we have developed descriptions of three hypothetical examples which provide a framework for the discussion in the rest of this report, as well as for subsequent work in the design of distributed operating systems. Readers familiar with the possible alternative requirements of applications may skip this chapter and proceed to Chapter 4 where reliability mechanisms are discussed.

We emphasize the aspects of these systems that are common to many application areas, rather than the specific designs. The three examples have been chosen to reflect a broad array of design constraints with respect to system performance, cost, reliability, and modularity. Finally, we have purposely assumed that each of the example applications is built on a distributed hardware base.

The most significant features of the systems differ along four dimensions: survivable vs. restartable operation; batch vs. interactive service demands; tightly vs. loosely coupled communication; and imposed priority vs. marketplace resource management.

Survivable vs. restartable operation. Reliability requirements vary in detail from one application to another, but can generally be grouped into two categories: survivable operation, which seeks to maintain continuous service in the face of component failures; and restartable operation, where some down time can be tolerated if a procedure exists for the smooth resumption of automatic processing when the component has been repaired or replaced. Examples of survivable systems can be found in military tactical systems, especially avionics [23, 71]. Survivable systems must employ redundancy of components to achieve continuous operation, and thus pay a clear cost penalty for reliability.

Often the additional cost of redundant processors and memory is not justified, and it is sufficient to accept some down time when a failure occurs. In this case it is particularly important that the system can be brought up-to-date automatically when repairs have been made, and that the system reaches a consistent state when it is restarted. Manual procedures may be necessary to support the application while the computer system is unavailable, and these manual procedures should be developed concurrently with the computer system software.

Batch vs. interactive service. Batch systems usually involve long processing runs against large, possibly distributed, databases. Batch runs are often scheduled in advance to run during periods of low system utilization (at night, on weekends), and are often periodic in nature (daily, weekly). Response time is not a critical constraint in batch processing and in the event of a system failure it is enough to restart a run from its most recent checkpoint when the system is again available. Interactive systems place much higher demands on response time. Some late responses are acceptable if the mean response time is small enough. Interactive requests are more frequent than batch and tend to be less predictable in terms of resource demands than batch requests. Because it is difficult to automatically select opportune checkpoints, recovery from failures is left largely in the hands of the users. At the minimum a positive indication of a failure should be supplied to the user, and some facility to limit the amount of work that can be lost due to a failure should be present.

Tightly vs. loosely coupled communication. Applications which are structured as multiple communicating processes are characterized as _tightly coupled_ if processes stop frequently, waiting for messages from other processes to proceed. Processors are tightly coupled if they can sustain a high bandwidth data transfer among themselves, with low startup delay. If a tightly coupled application is constructed on loosely coupled processors, the result may be poor performance.

For our purposes, we consider systems to be tightly coupled if they achieve a network throughput one to two orders of magnitude below the primary memory bandwidth of individual processors. By this criterion _local networks_ exemplified by the Ethernet [37], the DCS Ring [14] and Cambridge Ring [72] are considered tightly coupled systems. Local networks often employ radio-frequency carrier broadcast techniques, and at present are restricted to maximum diameters of about 3 kilometers. Local networks transmit messages with low error rates, permitting large block sizes and relatively simple error detection and retransmission protocols. The cost of a local network is a simple function of the number of attached units.

Loosely coupled systems usually rely on shared communication services, such as switched or dedicated telephone lines. The bandwidths obtainable are about three to six orders of magnitude below the primary memory bandwidth of contemporary processors; cost rises rapidly with bandwidth. The physical distance separating communicating units may be thousands of kilometers, or in the case of satellite links, tens of thousands. The cost of communication is a complex function of distance, data rate, volume of data, connection time, and other factors, and can be a substantial fraction of the total system operating cost. Shared networks have higher error rates than local networks, and this necessitates more elaborate error detection and handling schemes. Because shared communication networks serve many customers with different needs the interface protocols between hosts and the network may be complex (e.g., the X25 interface [54]).

Imposed priority vs. marketplace resource management. In many systems, particularly real-time and process control, administrative control over the utilization of resources by different tasks is essential. Process control systems are usually constructed with a rigid priority structure for dispatching time-critical tasks; these priorities reflect the damage which might result from failure to complete the tasks on schedule, as seen by the system administrator.

Other situations make it difficult or impossible to rank the importance of tasks competing for system resources (timesharing systems are a good example). In this case some form of marketplace economics may be appropriate, where system users pay for the consumption of resources in real or fictitious monetary units. A popular strategy is to treat the computing facility as a cost center, and attempt to maximize the utilization of system resources in order to minimize the cost per unit of computing to facility customers. This strategy is an extreme view, ignoring external priorities entirely for the common good.

3.2 Three Examples of Distributed Systems

In the sections below three examples of distributed systems are given in detail. They illustrate a broad range of military and commercial systems, present and anticipated, but should be regarded as completely hypothetical designs. The application area is briefly described for each system, and then a partial implementation is proposed. In subsequent chapters we will discuss many more implementation issues with regard to reliability, global scheduling and the support of distributed computations by host operating system primitives.

We caution the reader to consider these systems as representative, not as the sole systems to which our later results can be applied. Many of the details presented below are present only to enable a concrete discussion of the issues, and promote a sense of familiarity for the reader.

3.3 The Field Command Unit

3.3.1 Overview

3.3.1.1 General Description

The hypothetical Field Command Unit (FCU) is the closest general-purpose data processing element to actual military operations. Staffed by a Command Officer and a team of specialists in intelligence, logistics, air support, etc., the FCU commands and coordinates a combined air and ground force of several thousand people. The FCU is physically small and highly modular so that it can be airlifted and made operational within hours of a deployment order.

Modularity is a prime requirement of the FCU. The size of the combat force, equipment engaged, and distance from supply bases, among other factors, will affect the makeup of an FCU and its team. In order to achieve the high degree of adaptability desired, an FCU is structured as a local communication bus called the _spine_ and a collection of modules which attach to the spine and interact through it. A wide variety of modules are available for inclusion in a particular FCU; some are dedicated to a single function, for example high-speed signal analysis, while others are more general purpose and can be dynamically allocated by the FCU DOS.

A centralized warehouse facility located near an air base maintains an inventory of FCU components. Given the required capabilities of a specific FCU, the warehouse personnel use an automated configuration system to select the appropriate modules. For example, if the requirements include "participate in Autodin" the configuration system would select the necessary communication modules to interface the FCU to the Autodin network. Figure 3 shows the components of an FCU configured to permit packet radio communication, satellite communication, data encryption/ decryption, and to support a team of several specialists.

The modular approach exhibits flexibility not only in the range of function and capacity possible in an FCU, but also in the level of reliability attained. An FCU deployed for combat can be configured for high-reliability by including redundant components (including duplication of the spine itself) and adaptation of the DOS protocols to achieve fault tolerance and restartable operation. This mode of operation is produced by the configuration procedure at the warehousing facility. An FCU can be configured without the redundant components, for greater cost-effectiveness in systems used for training, maintenance, or installations not in the chain of command of combat forces.

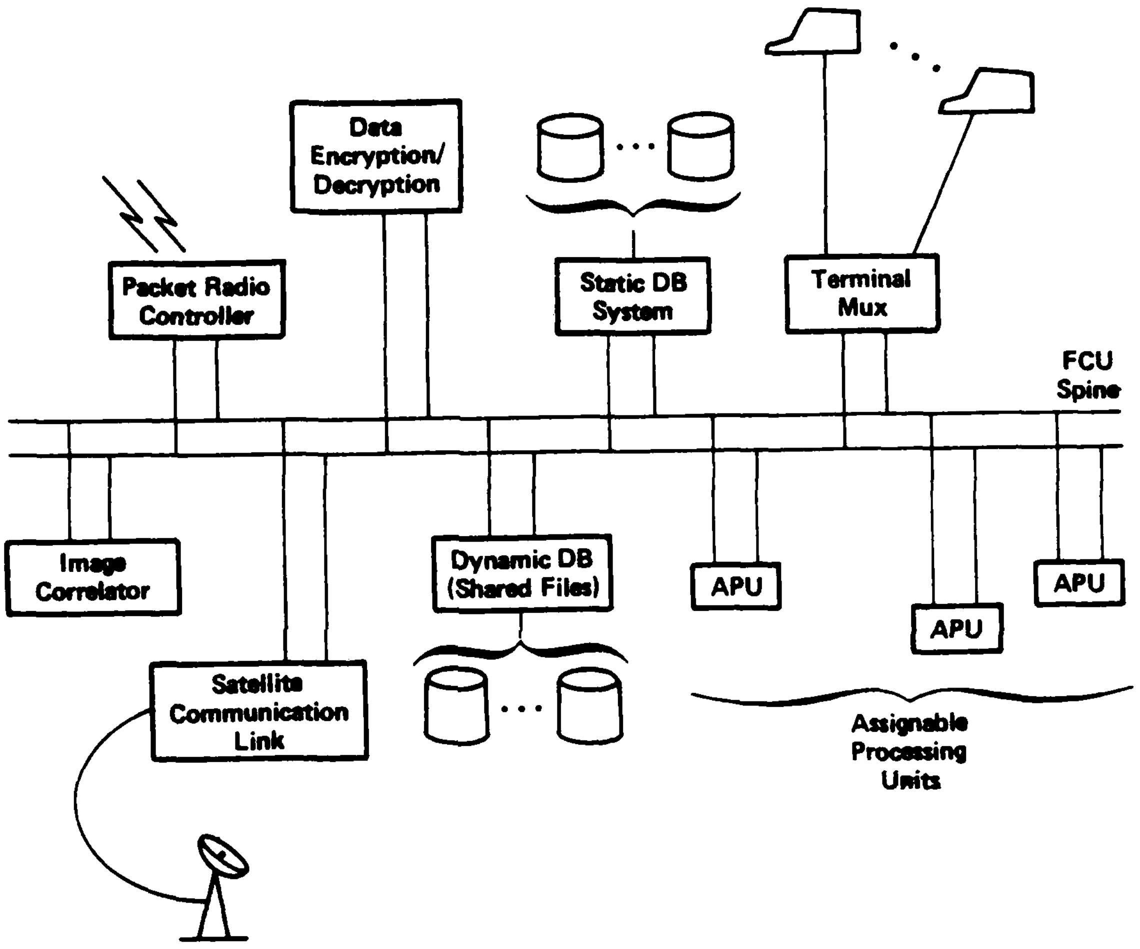

Data Encryption/ Decryption
Packet Radio Controller
Static DB System
Terminal Mux
FCU Spine
Image Correlator
Dynamic DB (Shared Files)
APU
APU
APU
Satellite Communication Link
Assignable Processing Units

Figure 3. Field Command Unit Block Diagram

3.3.1.2 Similarity to Other Systems

The FCU has some features in common with advanced C3 systems being developed for the Tactical Air Force. The concepts of modularity and standardization through the interface to a communications medium are the keys to lowered lifecycle costs and improved utility for systems in a tactical environment.

The structure of an FCU is reminiscent of several personal computer networks in use or under construction today at Xerox Palo Alto Research Center, Massachusetts Institute of Technology Laboratory for Computer Science, Carnegie Mellon University Computer Science Department and Bolt Beranek and Newman, Inc. It differs in that the unit of modularity of the FCU is rather small (e.g., processors and secondary storage units are separate modules) and the FCU spine is intended to be utilized heavily during normal system operation. A single FCU module is usually not capable of significant function independent of services provided by other FCU components.

3.3.2 System Topology

3.3.2.1 Hardware Components

The FCU spine is the common medium for communication among FCU modules. Every module has two spine interfaces, since the spine may be paired for high-reliability, and is capable of using either interface for any function. The bandwidth for messages through the spine is in the range of 1 to 10 Megabits/second. Data transfers are normally made between one sender and one receiver in packets of 1000 bits or less, although a mechanism is provided for broadcasts from one sender to many receivers. Because the spine is a shared resource, modules wishing to send may have to wait or send and retry if receipt is not acknowledged. The variance of data transfer times can become quite large if the bus is operated near its maximum bandwidth.

The FCU configuration in Figure 3 illustrates the use of Assignable Processing Units (APUs) to perform computing tasks on demand by the *team members at their graphic workstations*. An APU is a minicomputer executing approximately one million instructions per second and possessing upwards of 256 Kilobytes of local random access memory. The APU has no peripherals of its own, but reaches through the spine to use disk storage, printers, communication units, etc. The FCU DOS treats the APUs attached to the spine as a pool of available processors that may be assigned to tasks as the need arises. The assignments may be essentially static, persisting as long as the FCU operates, or may very brief and last only fractions of a second.

Disk storage in an FCU is supplied by _shared file modules_.

A shared file module is a small processor controlling one or more disk volumes; the processor responds to requests for data reads and data writes sent to it through the spine. Shared file modules may incorporate local intelligence for special functions, for example the automatic staging of files through communication links to larger data processing systems.

3.3.2.2 Task Assignments

Most computing tasks reside in APUs and use the services of other modules as necessary. The DOS has some resident code in every APU, although perhaps just enough to bootstrap additional code from shared file modules. Since APUs are identical any task can be run in any APU, and the DOS has full flexibility to perform dynamic task assignments if it chooses to do so.

Many of the devices we consider as FCU modules have little intelligence and must be controlled by software in an APU. We briefly describe two examples of this behavior.

Satellite Image Gathering. A subset of system modules consisting of an APU, the image correlator, the satellite communication link, and the static and dynamic database systems cooperate to discover meaningful events from satellite images. The APU, using the static database to retrieve geographical information and stored image templates, commands the satellite communication link to obtain an image of a given area. When the image arrives on the downlink, the APU transfers the image and template into the image correlator, sets the appropriate commands, and starts correlation. The APU detects the end of the correlation, checks the results for significant matches. If the correlation is negative the APU repeats the entire process after a suitable delay. If a positive match is found, the APU signals the Intelligence Officer who may then manually direct the display of images on a graphics screen along with the correlations and confidence estimates. The officer may then request direct control of the satellite imaging, change its frequency, or repeat correlation with altered parameters. During these steps, the APU uses the dynamic database to store a temporary (say, 24 hour) record of all images, in the event that details have passed unnoticed and the images are needed for later analysis.

Packet to Satellite Relay. Since this FCU has both packet radio and satellite communication modules, it may serve as a ground-space relay station. A subsystem composed of an APU, the packet radio controller, and the satellite communication link performs this task. Here the two communication modules are essentially passive components of the FCU, and all data transfer between them is done under control of a program running in the

APU. If this application and satellite imaging are running simultaneously, the issue of shared access to the satellite communication link must be addressed.

3.3.3 Application Requirements

3.3.3.1 Response Times

Because a wide range of applications can coexist on one FCU it is difficult to discuss achievable performance in general. The spine bandwidth does however pose a fundamental limitation on the number of applications that can be superimposed on one FCU. In configurations where this limit is approached, critical applications should try to use local storage and processing power rather than seeking them through the spine from other modules. For example, the operator's display station might incorporate storage for several screen images, that could be updated and selected for display by short commands from an APU.

3.3.3.2 Data Rates

The data rates present in a particular FCU will depend strongly on its configuration.

3.3.3.3 Reliability Requirements

When an FCU is configured for high-reliability operation it will be automatically restartable _without_ _repair_ in the event of any single failure. A single failure may cause a momentary interruption in some or all of the functions performed by the FCU, and the loss of some recently acquired data. A high-reliability FCU will automatically detect a failed component, isolate it from the remainder of the system, and restart in a well-defined state. Multiple failures can, of course, prevent a successful restart.

3.3.4 Appraisal

A high-reliability FCU is a restartable system that is permitted occasional lapses of a few seconds when failures occur.

The majority of the service demand in an FCU is interactive and arises either from operator commands or external communication activity. A few tasks may operate in the background with low priority for such operations as data logging and diagnostics.

The FCU modules are more tightly coupled than systems employing long communication lines, but less tightly coupled than multiprocessors with shared memory.

Priorities for many FCU tasks can be set at configuration time, because the complete set of functions to be provided by the FCU are explicit. The priorities may be contingent upon external factors, for example, the priority of weather information may rise during planning for an air strike.

3.4 Air Traffic Control

3.4.1 Overview

3.4.1.1 General Characteristics

The control of aircraft over a large area is an inherently distributed problem. The limited range of radar and the number of aircraft in flight at any moment mean that responsibility must be distributed among many human controllers at widely dispersed sites (in the United States, there are about twenty _en route_ control centers across the country). Close communication is needed between sites in order to smoothly accommodate the interchange of aircraft from one control sector to another, and to plan ahead for possible congestion as planes converge on one region. Long term planning is possible, since the majority of routine flights are regularly scheduled and the flight plans are known weeks in advance.

The primary goal of the system described in this example is the protection of the occupants of aircraft in flight, during the period after the tower relinquishes control on takeoff until the tower regains control immediately prior to landing. The system is thus an _en route_ air traffic control system, and must contend with three major threats to aircraft: hazardous weather, mid-air collision, and in-flight medical and mechanical emergencies. A secondary goal of the air traffic control system is the timely arrival of flights, conserving fuel and speeding the delivery of passengers and freight.

The example Air Traffic Control System (ATCS) is organized as a group of Flight Centers (FC's) spread across the country. An FC is responsible for controlling aircraft in a well-defined volume of airspace, its Flight Center Territory (FCT). The FCT's are each connected areas (actually volumes) that together exhaust the airspace controlled by the ATCS. In a low traffic area away from major airports an FCT may be large, while in a high traffic area an FCT may extend only slightly beyond the metropolitan boundaries. An FCT is further divided into _sectors_ by planar area and altitude.

An FC is staffed by _controllers_, the people with immediate responsibility for coordinating the movement of aircraft. Controllers are assigned in pairs to _workstations_ consisting of a large display screen, keyboard, graphical pointing device (a trackball), low-speed printer, and microphone-earphone headsets for the controllers. The headsets place controllers in direct voice communication with aircraft in their sectors and their counterparts at other FC's. A controller team is responsible for the control of aircraft in one sector of its FCT.

A controller usually concentrates attention on the display screen which provides four basic types of information:

- o A static map of the sector controlled by this workstation. Major landmarks are displayed and labelled (coastlines, bodies of water, mountain ridges, airports).
- o Symbols representing the aircraft in the sector (targets), marked with _tags_ indicating the flight number and velocity of the aircraft.
- o Outlines of weather fronts and atmospheric disturbances.
- o Blocks of text giving more detailed information on weather and flight plans.

The tagged targets and weather information are superimposed on the static map. The precise contents of the display can be influenced by commands typed at the keyboard, for instance to enhance or suppress weather information.

A frequent controller command is the _handoff_ performed whenever an aircraft crosses a sector boundary. The retiring controller types the handoff command and selects the target with the trackball. This causes the target to be intensified on the display of the accepting controller, who then types a complementary command and selects the target on the edge of his screen with the trackball. The retiring controller contacts the aircraft by voice radio and informs the aircraft of the radio frequency of the accepting controller. The radio operator contacts the new accepting controller to complete the handoff.

ATCS supports other activities of the controllers by maintaining the display with accurate position and velocity information for targets, weather information, and special status messages. ATCS propagates flight plans along the rcute of the aircraft in advance of the flight, so that controllers can familiarize themselves with incoming traffic and to implement a manual backup system. Several types of hazardous situations are continuously sought by ATCS (e.g., predicted dangerous proximity of aircraft, severe weather conditions, aircraft deviating from flight plans, or a congested volume of airspace) and announced to one or more controllers when detected. ATCS must also distinguish and track a number of <u>uncontrolled aircraft</u> in each sector, aircraft that are not participating in the ATCS control protocol.

At the administrative level, ATCS provides a means for dynamically altering the boundaries of a sector or an FCT. Boundary changes always involve a negotiation between at least two FC's and must insure that no aircraft participating in ATCS is without control at any time. Aircraft transferred from one controller to another as a result of a boundary change must be transferred by the standard handoff mechanism.

3.4.1.2 Similarity to Other Systems

The example is modelled upon the National Airspace System operated by the Federal Aviation Administration, although it differs in a few important respects. The architecture of ATCS proposed below is decentralized, while NAS relies on large computer mainframes at the control centers. It is not possible to dynamically reconfigure sector boundaries in NAS, nor does NAS support digitized speech channels on its data transmission lines. Overall, NAS shows the effects of evolutionary development while ATCS, a hypothetical system, is specified with fewer constraints for compatibility and gradual implementation.

The basic properties of ATCS are characteristic of air traffic control applications designed to handle routine traffic. The requirements for a military tactical system are somewhat different and this example is not directly applicable to that problem.

3.4.2 System Topology

3.4.2.1 Hardware Components

An FC is illustrated in Figure 4. The workstations are attached to small Workstation Processors (WP's); WP's are connected by a star-shaped network to a central hub and thereby to each other. Two of the nodes attached to the FC are gateways, processors that exist to facilitate communications between clusters. The gateways act as relays, forwarding intercluster messages along the appropriate lines.

The Workstation Processor is a small computer with local disk storage, a cluster interface, and special graphics hardware to drive the workstation display. The graphics hardware assumes the burden of display refresh so that the general purpose processing element of the WP may be devoted entirely to other tasks. A WP is also attached to a Coder/Decoder producing digitized speech from and translating digitized speech to the controllers' headsets at the workstation.

The star local network [49, 55] is chosen to interconnect workstations because of its reliability properties. No individual branch of the star is vital to the whole; even if the cable to one WP is severed the others will be able to communicate. The hub of the star is a critical component, but because it is relatively simple the hub can be constructed from high-reliability logic and a fault-tolerant design. At least two spare workstations (ordinarily used for training) are attached to each cluster. The star local network is capable of a maximum data rate of 10 megabits/second, while the intercluster lines operate at about 100 kilobits/second.

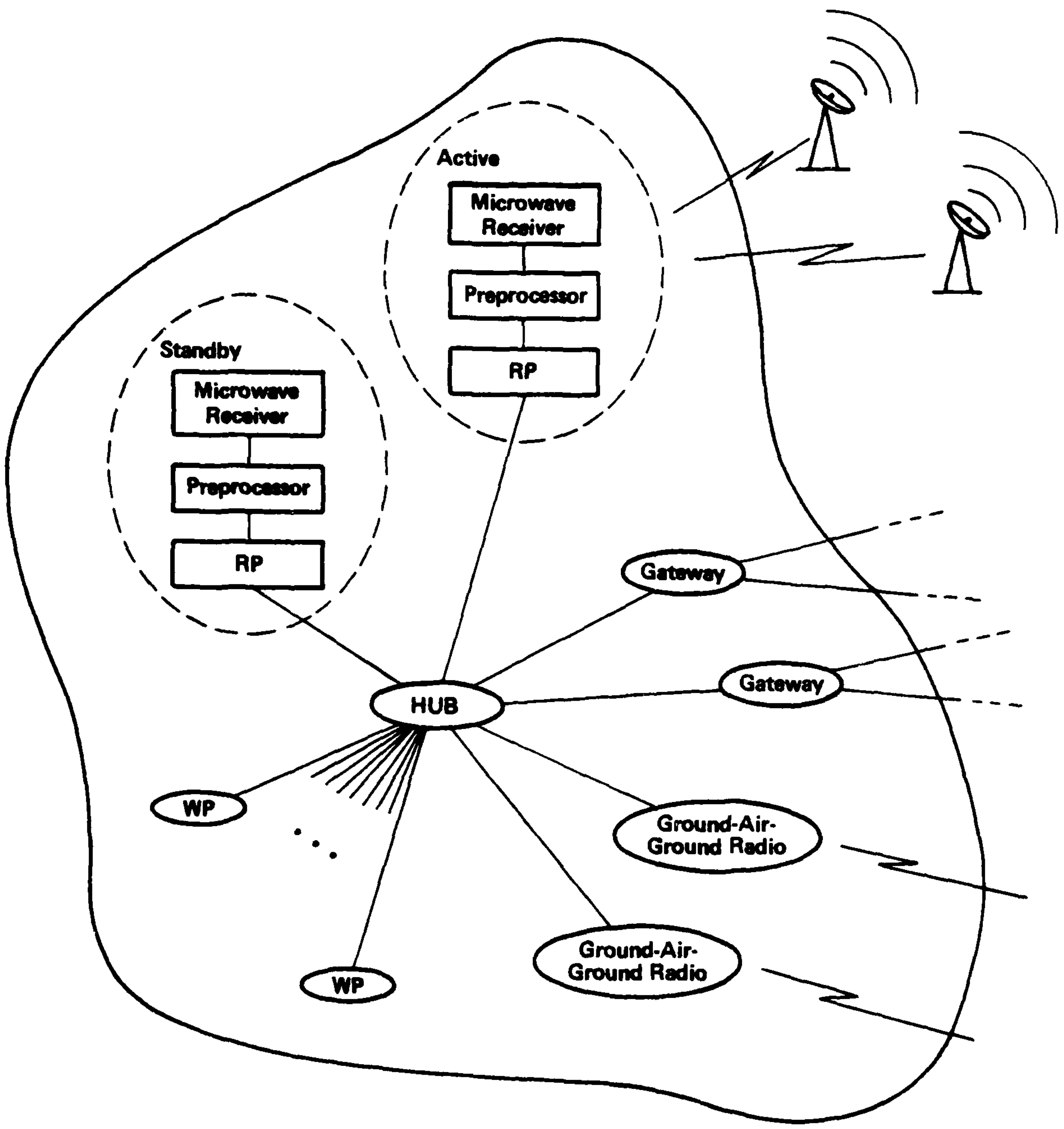


Figure 4. A Schematic of the Flight Center

Gateways are packet switches for intercluster messages. Each gateway is connected to two or more gateways in remote clusters; the two gateways in a cluster connect to at least four distinct remote clusters.

Radar information enters the system via microwave links or coaxial cable and is received by special-purpose data reduction hardware attached to the Radar Processor (RP). Two RP's are connected to each cluster, although only one is normally active while the other remains in standby mode. The active RP receives preprocessed image data and merges the information into one representation of the FCT. The RP utilizes last known position reports from the WP's to assist in target resolution, then supplies the present position for targets in a sector to that sector's workstation.

Also connected to the hub are at least two Ground-Air-Ground radio transceivers, which receive digitized speech from WP's and transmit audio signals to aircraft, and provide the reverse path.

3.4.2.2 Task Assignments

The WP at a workstation supports:

- o tracking and tagging the aircraft in the sector (computing velocity, for instance, from a sequence of prior positions and velocities);
- o updating the workstation display in such a way that all

tags remain legible irrespective of the relative positions of aircraft and landmarks;

- o receiving, storing, and forwarding the flight plans of aircraft passing through the sector;
- o the handoff protocol and a wide range of other less frequently requested keyboard commands;
- o digitized speech from the Coder through the star local network, and from the network through the Decoder to a controller;
- o low level activities such as sensing the trackball and keyboard;
- o the transmission of target velocity and position estimates to the active RP;
- o protocols to insure the reliable operation of the system as a whole, e.g., activating the standby RP if the active RP fails.

Gateways are just packet switches, their responsibility is limited to the reliable and speedy transmission of intercluster packets.

The radar processors receive preprocessed radar image data from their front ends, merge several images and target reports from WP's, and resolve new target positions which are reported back to the WP's for display. Because this must be done in real time the computing burden on the RP is substantial.

The hub is not a programmable computing element but rather the switching hardware that binds the local cluster of WP's Gateways, RP's, and Ground-Air-Ground radio together. Basically a passive component, the hub only receives and repeats signals with extremely high reliability.

3.4.3 Application Requirements

3.4.3.1 Response Times

The most severe requirement is for real time updating of WP displays. The positions of all targets must be recomputed and displayed at least every two seconds (the time between radar scans). The tag information (flight number, altitude, and velocity) should should be updated at about the same rate.

The WP must be even more responsive when the trackball is used to select a target. No visible delay should be present between the motion of the trackball and the motion of the crosshairs on the screen. The controller's digitized speech messages must also be processed without delay.

Messages in a handoff dialogue should be transmitted with a delay of less than 4 or 5 seconds.

Other routine services can be performed more slowly. For example, weather information and flight plans can percolate through ATCS with propagation times of several minutes. Reconfiguration will be slower still, taking tens of minutes to adjust sector assignments and notify the affected controllers.

3.4.3.2 Data Rates

Because the active RP must receive position and velocity reports for targets from all sectors every two seconds, and because the RP generates position updates at the same rate, this activity represents a major load on the ATCS local network. A full report on all aircraft in the sector might range from 4K to 40K bits per workstation, for a worst case of 40 kilobits/second/ workstation. Digitized speech between controllers and aircraft generates significant traffic in relatively short bursts. A telephone quality speech channel can be operated at 16 kilobits/second during continuous speech; 20 controllers speaking simultaneously would produce a traffic volume of 320 kilobits/second.

It is highly desirable to maintain the local network utilization at a level well below 50%. This is necessary to reduce the variance on transmission time, for digitized speech packets especially. The local network bandwidth of 10 megabits/ second (within the capabilities of current local network technology) will permit at least 20 workstations to be assembled in one FC before the network utilization rises above 10%.

3.4.3.3 Reliability

Continuous operation is paramount in air traffic control where lives are at stake. The example ATCS is designed to make the probability of a disruptive failure extremely small. The system would be highly redundant and programmed for fault-tolerant operation.

In the event of catastrophic failure, manual operation could be initiated using printed flight plans at each workstation and a direct video image supplied to the workstations from the radar equipment. Separate cabling could be used to carry direct video and headsets can be connected to telephone lines, for complete diversification.

The WP's continuously monitor the status of their colleagues and the RP's. If a WP fails, one of the standby WP's will be converted to active status, loaded with the data pertinent to the sector of the failed WP, and started. A similar procedure is followed if the active RP fails, but an RP's state information is more volatile and need not be replaced before starting. When the RP is switched in, all of the WP's must be notified of the change.

3.4.4 Appraisal

The ATCS is a survivable system requiring a large investment in redundant equipment and fault-tolerant hardware. Redundancy must be present in all of the related subsystems including radar, radio, telephones, and power supplies.

The system is fully interactive with no substantial batch processing components.

The WP's and RP's within a cluster are tightly coupled physically and logically. They are connected by a high bandwidth local bus and utilize a large fraction of the available bandwidth continuously. Intercluster coupling is less tight, and could be accommodated by slow lines if the digitized speech channels were replaced by external telephone lines.

The WP's operate independently and with equal priorities for access to the local network. Wherever possible, priority is given to digitized speech packets because of their real-time nature, both with regard to tasking priority within the WP's and Gateways, and for access priority to the local network. The messages exchanged between the RP and the WP's can be time multiplexed to avoid collisions.

3.5 A Military Message System

3.5.1 Overview

3.5.1.1 General Characteristics

A military command center could be served by the military Message System (MS) developed in this example. The message system connects many command centers and facilitates both inter- and intra-center message traffic. The system helps the command center staff compose, send, read, and file messages in a way that meshes smoothly with normal routine. Archival copies of messages sent and received are made automatically, and a database management component retrieves old messages from the archives.

The users are divided into _Directorates_ each commanded by a Director, who is assisted by a Deputy Director. Incoming messages arrive at the message system node (MSN), are briefly reviewed by the communication center staff, and then forwarded to the correct Directorate. (Because messages often arrive without a specific address and must be delivered on the basis of subject matter, this distribution step cannot be fully automated.) The Deputy Director of the Directorate receiving the message is then responsible for further distribution to staff members or the Director. As messages pass through the communications center they are assigned a unique message-ID and copied into the archives. Outgoing messages are stamped with a unique ID and archived when they are sent, but need not be reviewed.

Messages are marked with priority level (one of Flash, Immediate, Situation, Routine) and security level (one of EyesOnly, Secret, Unrestricted). The procedures followed when messages are sent and received depend on the priority and security levels. For example, the arrival of a Flash message will cause immediate notification regardless of work in progress by the recipient. If the recipient fails to acknowledge receipt promptly the message will be forwarded to another party (if it is not EyesOnly) until responsibility for the message is accepted. EyesOnly messages are never revealed to anyone except the addressee (they are addressed to individuals, rather than roles) and are never archived.

3.5.1.2 Similarity to Other Systems

This example is based loosely on a study of CINCPAC done by Mitre Corporation [19]. The message handling procedures described in that report were largely manual. The Military Message Experiment (MME) [62] was a subsequent effort to test automated message handling at CINCPAC.

The MME may also be viewed as an outgrowth of the numerous message systems developed on the ARPANET. A representative of these is the Hermes message system [42].

3.5.2 System Topology

3.5.2.1 Hardware Components

A single node of MS such as the one installed at the military command center consists of four computers, depicted in Figure 5. Two of the computers are relatively small and simple _terminal concentrators_. Every terminal at the node is attached to one of the concentrators. If a concentrator fails, half of the terminals will be cut off. Concentrators are small and simple enough to be built with high-reliability architectures, and spares can be held in reserve. Thus even though the consequences of a concentrator failure are severe, it is expected to occur very infrequently and can be easily repaired.

The concentrators are connected to two larger _server_ machines. Each concentrator is connected to both machines and either server is capable of sustaining the full computing load of MS at the command center. A failure of the left or right server will cause the other machine to automatically assume the computing tasks of all online users, with no loss of input data and perhaps only a slight pause (less than 5 seconds) in response.

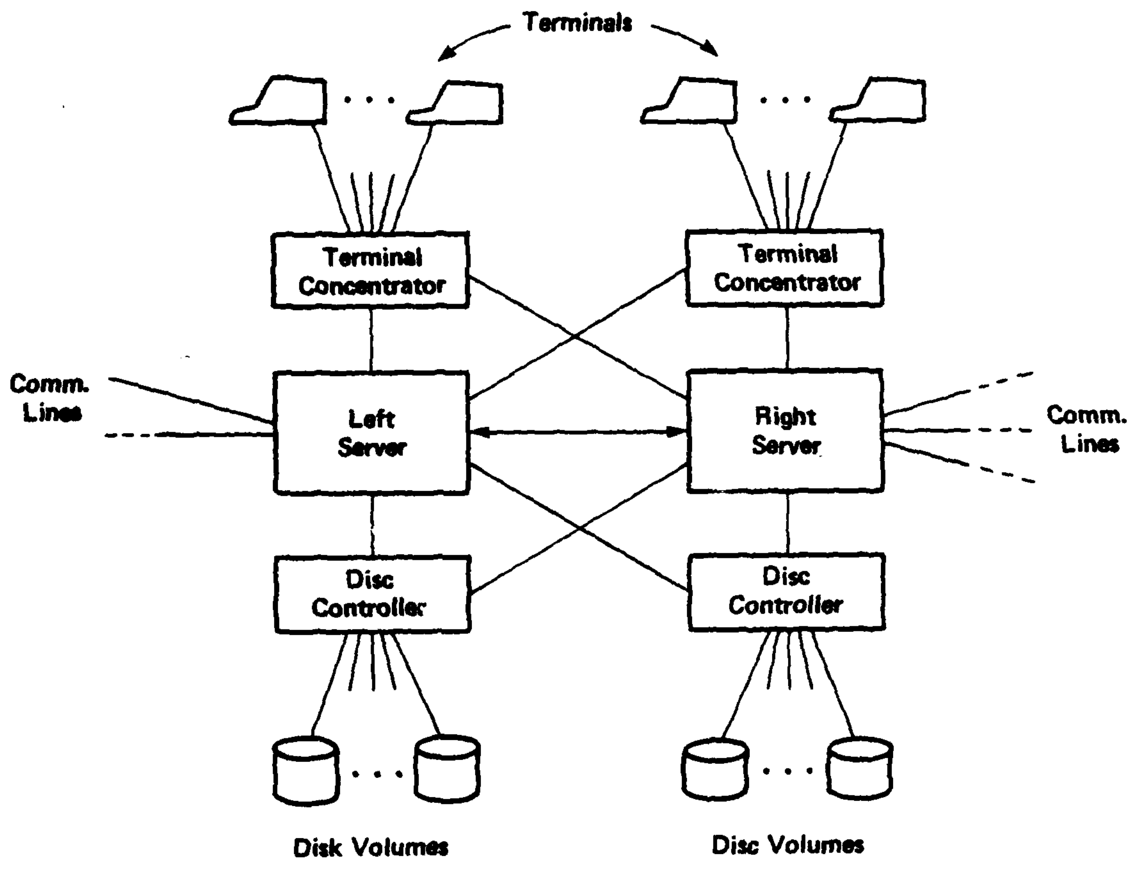


Figure 5. A Message System Node

The servers cooperate to insure fully redundant storage of all data in the system, via replicated disk controllers and disk volumes. The failure of one disk volume will cause all queries and updates to be directed to its dual until the failed unit is repaired or replaced. A newly introduced volume will be brought online and made up-to-date by copying information to it from its dual.

Communication lines to other MSNs are split between the left and right servers so that failure of one server will not isolate the MSN from the network. Each server should be connected to two distinct MSN systems to reduce vulnerability to communication line failures. The communication lines can transfer data at 250 Kilobaud through a Direct Memory Access (DMA) interface to the left or right server's memory. The servers act as packet switches, breaking down long messages into packets of 4000 bits or less. A MSN may introduce new packets into the network, extract packets addressed to it, or forward packets addressed to other MSNs.

3.5.2.2 Task Assignments

An MSN furnishes services to local users and acts as a packet switch for through traffic. The left and right servers must be able to assume all of the processing tasks of the node if necessary.

Most of the system computing capacity will be devoted to two tasks: interrupt-driven packet switching, and the communication between servers needed to insure survivable operation. The concentrators can be depended upon to preserve a small amount of the recent communication history between servers and terminals, which can be used by the surviving server in the event of failure to resume the dialogue properly.

3.5.3 Application Requirements

3.5.3.1 Response Times

Because all access to the system is interactive it is desirable for all response times to be as small as practical. Priorities for tasks can be established, though, based on four classes of user expectations:

1. **Coordinated hand-eye movements.** In situations requiring coordinated hand-eye movements, response times should be shorter than the minimum human hand-eye response time (about 0.2 seconds). For example, the delay between pressing a cursor positioning key and movement of the cursor on the screen is in this class.

2. **Immediate actions.** Actions perceived by the user as being "local" or "personal" should occur very quickly, ordinarily with delays of 2 seconds or less. For example, "turning a page" or viewing the next screenful of text in a message is an example in this class.

3. **Significant computation.** If the user initiates a "significant computation," that is, a task he understands will accomplish a substantial search or

reorganization of information, he may be prepared to wait many seconds before proceeding. Examples in this class include text formatting and complex database queries.

4. **Remote Actions.** If the user perceives a command as having effects at a distance he may accept delays of minutes to hours for a response. Message transmitted to remote MS users are the principal example of this class, and network status inquiries might be another.

Response requirements may also be strongly affected by two external factors, message priorities and hardware interrupt processing deadlines. Some hardware devices will impose scheduling deadlines for associated server tasks; DMA communication interrupts are probably the most important instance of this. Interrupts must be processed and the DMA interface reset to avoid the loss of the next incoming packet.

3.5.3.2 Data Rates

When both processors are functioning, an MS node should be capable of a packet switching throughput approaching the data rate of its communication lines, assuming half are transmitting at the maximum rate and the others are receiving the forwarded packets.

If simple text-only display terminals are used, data rates for terminal I/O can be estimated from these requirements, a knowledge of the user interface, and statistical information on the frequency with which operators invoke various commands.

We assume that a typical MSN is constructed around servers with approximately the computing power of the PDP 11/70, and can support about 16 users.

3.5.3.3 Reliability Requirements

The failure of any single processing or storage element (server, terminal concentrator, disk controller, or disk volume) should not disable the system or disconnect it from other nodes. The failure of a terminal concentrator will disrupt communication with about half of the terminals, but should not affect the remainder. A failure in any component except the concentrators should not interrupt service to users _in any way_ except for the slight delay during which the system reconfigures itself. No input typed by the user should be lost if a server, disk controller, or disk volume fails.

To minimize the time during which a node is vulnerable to a single failure (i.e., after it has already experienced a single failure) the surviving components should attempt to isolate the failure and notify a system operator. When repairs or replacements have been made, the system should automatically reincorporate the element to reach a survivable operating mode within a few minutes.

3.5.4 Appraisal

On the spectrum of reliability, MS is a survivable system with severe reliability requirements. Such requests force fully redundant storage and processing components to be included in the MSN, and substantial computing resources to be devoted to checkpointing, survivable protocols, and database synchronization.

MS is primarily an interactive system. A node may perform some services in the background, such as file archiving and mail delivery to users not logged in, but most of the system resources will be consumed by on-demand activities.

Mail systems are inherently oriented towards loosely-coupled communication lines. Delays of a few minutes for message transmission are usually not significant, and a priority structure in the application provides a means for achieving short transmission times when needed.

The resource management within MS can be organized on a fixed, priority basis. The roles of system users are known in advance, and resources can be allocated in accord with these constraints. When priority is not critical, users can be served in an approximation to processor-sharing to reduce the average response time.

4. RELIABILITY MECHANISMS AND THEIR APPLICATION TO DISTRIBUTED SYSTEMS

4.1 Introduction

One key attribute of any usable computer system (or, for that matter, any other type of system) is the ability to rely on the system to correctly execute a set of commands given to it. Systems that are unreliable tend not to be used because they cause more work than they save. In Phase I of the DOS Design Study, we have investigated several different topics relating to reliability mechanisms for distributed computer systems:

1. **Application Requirements:** What are the requirements of actual applications? We have identified the demands several different classes of applications place on *mechanisms to insure* reliable operation. For example, some applications in life-controlling situations require absolute, uninterrupted service for long periods of time. Others, such as inventory control, also require reliable operation, but can tolerate periods of outage and backups to previous consistent states of system operation. Requirements of applications have been developed in Chapter 3.

2. **Generic Reliability Mechanisms:** Many different mechanisms for providing reliable operation have been developed. Groups of these approaches are really slightly different variations on one generic idea. An example of such a generic mechanism is the checkpoint/restart mechanism: many different variations and uses of checkpointing and restoring consistent system states have been developed.

3. **Specific Uni-Processor, Multi-Processor and Distributed Processor Reliability Mechanisms:** Advantage is taken of the relationships between the components of a system in the design of reliability mechanisms. Relationships that are exploited include: the physical proximity of

system components (e.g. processors and memories), the presence of inexpensive, redundant resources, and the ability halt concurrent activity while a consistent state is established.

4. **New Capabilities and Requirements of Distributed Systems:** Distributed systems represent both an opportunity and a challenge for providing reliable operation. The opportunity comes in the form of autonomous operation of the independent components that make up the distributed system. The challenge arises because of the difficulty of coordinating the operation of each of the autonomous components.

In this Chapter, the results of the reliability section of the Study are presented. In the next section, the nature of the mechanisms we investigated is described. Section 4.3 defines the terms associated with reliability mechanisms used in this Chapter. Next, the general reliability requirements of application programs and the corresponding goals of the mechanisms studied are presented. In Section 4.5 a survey of current techniques for providing reliable operation in computer systems is presented. The next section describes several different systems in which a number of different distinct reliability measures have been integrated together into a unified system. The final two sections address the new reliability requirements and capabilities of distributed systems and areas that need further study.

In addition, the discussion of reliability draws on the

<u>capabilities</u> motivated by Chapter 2 and the <u>requirements</u> motivated by Chapter 3. The results of this Chapter will be used in Phase II of this study in the development of desired features of both a Constituent Operating System and a Distributed Operating System.

4.2 Focus of Study

The primary focus in this part of the study has been on reliability mechanisms that are applicable to distributed systems. Certain mechanisms, while interesting in general, will not be successful in an environment where communication between the parts is significantly slower then the access time of one part to its local data. There is limited interest in mechanisms that either require close synchronization between components or large amounts of data to be transferred between components. In addition, we have emphasized mechanisms that work at the highest levels of system abstractions to the exclusion of very low level mechanisms. As an example, we have avoided studying reliability mechanisms such as parity checking and data consistency across subroutine calling boundaries in favor of larger granularity mechanisms such as atomic transactions (a form of short lived process) and multiple component interactions.

Finally, we have explored in greatest detail mechanisms that fit traditional models of computing. The main reason for this perspective is that to a great extent, existing systems and applications utilizing the models suggested by existing systems will be used as building blocks for future distributed systems.

4.3 Definitions

In the literature discussing reliability, there are many different interpretations given to several commonly used terms. Additionally, the meaning of the reliability of a system is often confusing or unstated. To avoid similar problems in this report, we present a set of definitions of terms (drawn largely from [48]) and also characterize what we mean by the reliability of a system.

4.3.1 Terms

The reliability of a system must be stated in terms of several different interactions:

- **System:** A system is a set of components acting together to perform a service. The components may be viewed at one time as atomic objects and at another, as systems themselves. The interaction between the components is the algorithm of the system.

- **Reliability:** The reliability of a system is the total measure of success of the system carrying out its specified service. This measure is taken after all the reliability mechanisms of the algorithm have been exercised.

- **Fault:** A fault is the physical or logical incident that causes the algorithm of a system to take an incorrect path, make an incorrect decision or store an incorrect value of data. A fault is the incorrect event while an error (see below) is the effect of that event.

- **Error:** An error is the effect of a fault on the data bases or interactions between components of a system. It is proper to speak of both fault repair and error correction; the former is the act of eliminating the source of faults while the latter involves eliminating the effect of a fault on the state of a system. We are concerned with errors and error correction when we attempt to restore the state of a data base to a previously known correct state. When we reconfigure the components used to make up a system, we are aware of faults and performing fault repair.

- **Error detection:** Three classical aspects of reliability mechanisms are error detection, error isolation and error recovery. Error detection refers to activities aimed at discovering the introduction of errors into the state of a system. The various techniques used include explicit checks of the consistency of results, error correcting codes, and timeouts.

- **Error isolation:** Error isolation refers to efforts at keeping the impact of errors isolated from parts of the system state known to be correct. Error isolation is usually an activity that requires planning ahead to make sure that partially developed, unverified results are kept separate from the main system state. An example of an error isolation mechanism is an intentions lists (see Section 4.5.4.5).

- **Error recovery:** Error recovery refers to those actions aimed at removing errors from the state of a system. The extent to which error isolation has been effective influences the difficulty and effectiveness of error recovery operations. Some error recovery techniques will establish a system state which is equivalent to the state that would have been in effect if the error had not been introduced by a fault. Others will only be able to set the system to a legal or consistent state -- one from which further system operation may proceed.

- **Error tolerant system:** An error tolerant system is one which makes use of error detection, isolation and recovery mechanisms to compensate for the effect of a fault introducing an error into the state of a system.

- **Fault tolerant system:** A fault tolerant system is one which is capable of being reconfigured so that the source of a fault is eliminated. Error tolerant systems are not necessarily fault tolerant and vice versa. However, if a reliable system (see below) is to avoid repeatedly spending resources on recoverying from errors, then it must be both fault tolerant and error tolerant.

- **Reliable System:** A reliable system incorporates both error tolerant and fault tolerant mechanisms to achieve, with high probability, the stated service of a system. Note that the reliability of a system is a probabilistic measure. Error and fault tolerant mechanisms can only reduce the probability that a system will fail, never completely eliminate the possibility of failure.

4.3.2 The Purpose of Reliability Mechanisms

With these defined terms as background, the purpose of reliability mechanisms is to keep a system performing the service it is supposed to provide in the presence of faults and the errors that faults induce in a system. There are many different techniques used as reliability mechanisms and the success which these mechanisms achieve also varies widely. Some very simple, inexpensive techniques may yield a high probability of success. Going the rest of the distance, achieving even higher probabilities of correct operation, can be costly and complex.

4.4 General Requirements and Capabilities of Reliability Mechanisms in Distributed Systems

As discussed in Chapter 3, different applications have varying reliability requirements and make different demands on the reliability mechanisms of an operating system. Likewise, different techniques for providing reliable operation have varying strengths and weaknesses, costs and complexities. In this section we characterize both the requirements of applications and the capabilities of various reliability mechanisms. First, however, one qualifying observation on the probabilistic success of any reliability mechanism.

4.4.1 Probabilistic Success of Operation

There is no such thing as a completely reliable system. Whenever there is a source of data there is always the possibility, however small, of a fault occurring which will introduce an error into a data base. When that fault occurs in the very mechanism that was supposed to compensate for the effect of faults, then the system will fail. This does not mean that it is impossible to developing reliable systems. Rather, the reliability of a system must be stated probabilistically -- e.g. a system will perform its service correctly with probability 0.993 .

In their paper on crash recovery in a distributed file system, Lampson and Sturgis [32] present a strategy for recovering from failures of components of a file system (e.g., disks, processors and communication links). They point out that the correctness of any algorithm is based on a formal model of the behavior of the devices used by the algorithm. Since there is no way of proving that the model and the physical device are identical, the best that can be done is to say that the physical device and the model are identical with probability (1-P) and that P is small enough. The value of P can only be determined by actual experience with the devices.

Lampson and Sturgis further characterize the probability of their mechanisms failing by dividing the events that occur in the formal model of devices into two categories: <u>desired</u> and <u>undesired</u> events (or, in our terminology, faults). In a fault-free system, only desired events will occur. They further divide undesired events into two categories: <u>expected</u> and <u>unexpected</u>. Error recovery mechanisms are prepared to deal with a bounded number of expected, undesired events and no unexpected, undesired events. The probability P is just the probability that an unexpected, undesired event occurs. While designers of reliability mechanisms attempt to account for all undesired events, there is always a possibility that an unexpected, undesired event will occur. Careful thought and experience with the nature of undesired events of a given device can, however, yield an acceptable probability of reliable operation. Similar results and analysis are reported in the context of the Tandem Guardian Operating System [2].

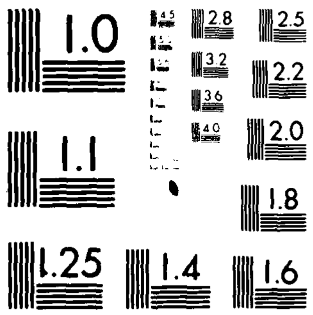
4.4.2 Application Requirements and Mechanism Capabilities

There are two general requirements of distributed applications: maintaining the consistency of the state of a system and continuing the operation of a system, both in the presence of faults. Maintaining consistency refers to preserving the integrity of the data bases that define the state of a system. Without such consistency, programs that rely on the data base cannot make sense out of its contents. Continuing the operation of a system involves seeking alternative means of accomplishing a service when a component of the system is no longer performing its service. Different applications have varying needs for continued operation ranging from completely continuous, fully functional operation to severely degraded, but partially functional, operation.

An application program whose data bases are kept consistent may still experience the effect of faults. Thus, there is need

AD-A113 172 BOLT BERANEK AND NEWMAN INC CAMBRIDGE MA F/G 9/2
DISTRIBUTED OPERATING SYSTEM DESIGN STUDY. VOLUME I.(U)
JAN 82 H C FORSDICK, W I MACGREGOR F30602-79-C-0193
UNCLASSIFIED BBN-4674-VOL-1 RADC-TR-81-384-VOL-1 NL

2 of 3
AD
A113 172



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS 1963 A

for both data base consistency preservation mechanisms and mechanisms for insuring the continued operation of an application. Of the two, an application's requirement on consistency maintenance has priority over its requirement for continued operation. This is because there can be no hope of invoking recovery mechanisms if the foundation of the system is uninterpretable.

With these general application _requirements_ as background, the _capabilities_ of reliability mechanisms can be described in a similar manner in terms of consistency maintenance and continued operation of a system. Such general capabilities are valid for both centralized and distributed systems. While it is rare that one reliability mechanism can be classified as being motivated by just one goal, examples of mechanisms which are largely inspired by consistency maintenance or service continuation do exist.

The essential goal of consistency maintenance is the integrity of stored data. This can take several forms:

o State Consistency: Logically correct states of a system are not made incorrect due to faults. The goal of some reliability mechanisms is to insure that the correct logic of a program is not made incorrect by being interrupted, interfered with or partially or incorrectly completed.

  Application Requirement Example: One of the most elementary requirements of almost all applications is that the data bases of the application be kept consistent so that the programs of the application can continue to operate.

Mechanism Capability Example: An atomic update[2] is a modification to a data base that cannot be divided into multiple fault sensitive parts. An atomic update either succeeds entirely or fails completely -- completely in the sense that the data base state is the same as it was before the atomic update was started. There is no in-between state established where part of an update is applied and part is not.

o **Permanence of Operations:** If a program performs an operation, then the effect of the operation (updates to the data bases of the system) is permanent. It is desirable to say that after a given point, a given update to a data base will be insulated from the effects of faults.

Application Requirement Example: A predominant mode of using computer systems is to record information. Airline reservation systems, warehouse inventory control systems, electronic message systems all rely on computer systems to take in information and to not loose that information.

Mechanism Capability Example: A common version of this type of reliability mechanism involves moving a piece of a system's state from volatile to non-volatile storage. The act of such a transfer is to lower the probability that a fault will cause the data to be lost. This is useful, for example in the programming of a server application program which accepts request, immediately delivers an acknowledgement for a request and eventually delivers the response to a request. Before the request is acknowledged, a record of that request must be moved to non-volatile storage so that it will not be forgotten

[2]Atomic updates are used for several different purposes besides achieving reliable operation. Another goal is to use them in conjunction with atomic transactions as a means of enforcing exclusive access to a data base by one of several competing processes.

due to a fault in the processor or volatile memory.

The second major capability or goal of reliability mechanisms is aiding in the continued operation of a system. Several different types of mechanisms fall into this category, including exception handling, persistence mechanisms, alternative sources of service and reconfiguration strategies. Four general categories can be described:

o **High Availability**: The goal of mechanisms for achieving high availability is to help provide a service that is available with probability Q, where Q is close to 1 and in any case to limit the duration of outage to some period of time **delta t**, where **delta t** is determined by the characteristics of the application.

  Application Requirement Example: Every system has some need for availability, so the magnitude of Q and the value of **delta t** determines the degree of high availability required by an application. For example, a life support application might require the utmost in high availability with a very high Q and a very short delta t, while an electronic telephone switching system would require high availability, but with much less urgency.

  Mechanism Capability Example: Most applications that require very high availability make use of duplicate processor and memory resources that are able to pick up the computing of the application in the event of a failure. Systems such as #1 ESS [67] and the Tandem Guardian Operating System [2] utilize redundant processors, memory and devices that keep track of all system data bases so that in the event of a fault, the failed component can be dropped [illegible] the computation proceed with the back-up component. Quite often techniques for providing high availability are specialized for the application in question, taking advantage of well understood characteristics and requirements of the application. Some very critical tasks of an application could have redundant resources allocated to them, while other, less critical tasks would not need to be supported as effectively.

- **Fail Soft:** Fail soft operation admits the possibility that the occurrence of a fault will change the characteristics of an application program. Changes can be in both degraded performance and degraded functionality. The important feature, however, is that a fault does not cause all application tasks (including unfailed components) to cease operation. The errors caused by faults must be isolated so that their presence does not cause other correctly operating components to fail also. In addition, applications that must exhibit fail soft behavior must be structured so that no single component is critical to the overall operation of the application.

  Application Requirement Example: Almost any application which is routinely used to support critical activities has a requirement for fail soft behavior. In a data base management application, the inability to access part of a distributed data base, should not preclude satisfying queries that do not involve the unavailable portion. For a status monitoring application, the failure to receive information over a communication path about one monitored event should not impact the monitoring of other events. In addition, alternative means of determining the status of the event could be used to compensate for the failed path.

  Mechanism Capability Example: In the ARPANet, the algorithms used to route messages between the message switching nodes (IMPs) are designed to compensate for failed nodes. In event that a node along the path of a message fails, the dynamic routing algorithm chooses an alternative path that will bypass the failed node. While this results in somewhat degraded service, the network still manages to get messages from one point to another.

- **Recovery of Partial Results:** In an application which involves expensive calculations, there is a great desire to avoid recomputing such calculations in the event of a fault. This is not so much motivated by the criticality of the computation, but rather by the sheer size of the task.

  Application Requirement Example: A computation which runs on a computer system for an extended period of time (say, one hour or more) requires some form of partial results recovery mechanism. An example of such a program is a weather forecasting calculation: there are large amounts of computing to be done to perform the algorithms of weather forecasting and in addition, there is no time to re-run the entire calculation before the next period of weather forecasting occurs.

Mechanism Capability Example: Checkpoint/Restart mechanisms are the most familiar versions of this form of reliability mechanism. The basic idea is to store the state of a long, expensive computation at several intermediate points (checkpoints) of the computation so that the occurrence of a fault will require recomputation only from the most recent checkpoint, not from the beginning of the entire computation.

- **Task Completion:** In some situations, there is a desire to complete a task regardless of the method or time used to accomplish this. The fact that the task was not completed as soon as possible is not as important as the ultimate success in completing the task.

  Application Requirement Example: The electronic mail system of the ARPANET illustrates this requirement well. Between the time a message is composed and the time it is delivered to its recipient, many different types of faults can occur: the sending or receiving host can crash, one of the nodes in the the communications network can stop working and messages can get lost. The only requirement is that the message get from the sender to the recipient.

  Mechanism Capability Example: A set of techniques collected under the term Persistence can be used to achieve ultimate completion of a task. One such technique is the following general framework to be used in application programs:

  1. Attempt a task.

  2. Wait for a positive response indicating the completion of the task.

  3. If a positive response is not received after a period of time, abort the operation, remember the need to perform the task and try again after a

waiting period of time.

This is a very simple technique and will only handle certain types of reasons for not completing tasks. However, in many cases (as in the ARPANet electronic mail example) this approach is satisfactory.

In addition to the above capabilities of centralized or distributed reliability mechanisms there are several attributes unique to distributed systems which enable additional techniques for reliability mechanisms to be used. Implicit in many of the classes of mechanisms stated above is the assumption that some parts of a system may fail and others will still operate correctly. For centralized systems (e.g., an application program written to operate on a single host computer system), this is a questionable assumption: usually, the system *runs or it doesn't* -- there are no intermediate states. With distributed systems, there can be true autonomy. When separate hosts are physically separated, the failure of one machine does not impact the operation of another. One of the impacts of a distributed architecture on potential reliability mechanisms is that the effectiveness of mechanisms is enhanced by the true autonomy of the multiple components that make up a distributed system.

Another characteristic of a distributed architecture, the greater emphasis on decoupled systems, also enhances the potential for reliable operation of application programs.

Implicit in most of the types of mechanisms discussed above is the separation of an application into multiple parts. This is primarily to achieve multiple autonomous parts in an application and also multiple parts which may be flexibly connected together to create one application. With centralized systems, it is difficult to avoid building couplings that take advantage of the central machine upon which the application runs. These couplings tend to be rigid and difficult to break and reestablish in a different configuration.

The result of a distributed architecture is to make many of the techniques that have been suggested for building reliable systems on centralized systems much more believable. In addition, new techniques that take advantage of the true autonomy of the components of a distributed system can be developed.

4.5 State of Current Technologies for Providing Reliable Operation

4.5.1 Introduction

The subject of reliability and mechanisms to achieve reliable systems is given excellent treatment in published literature. In the course of the study, we discovered three, rather different, types of contributions:

- **General Approaches, Frameworks and Surveys**: Several excellent survey papers exist that describe general views of the field of reliability, the nature of faults, and mechanisms for dealing with the effect of faults on the operation of a system. Anderson, Lee and Shrivastava [1], and Verhofstad [69] have each written survey articles which are good reviews of the various terms and techniques that have been used in general purpose operating systems and data management systems, respectively.

- ***Isolated Proposals for Specific Mechanisms***: The second type of contribution are specific mechanisms for attacking specific problems in computing systems. Many different mechanisms have been proposed and there is much overlap between many of the proposals. Lomet's original work on atomic actions [36], Montgomery's proposal for using multi-valued objects to implement atomic actions [41], Lampson and Sturgis's proposal for using atomic actions in a crash recovery mechanism [32] all address the issue of atomic actions, but each addresses a different aspect of the mechanism and use it in slightly incompatible circumstances to solve different problems. Such proposals of isolated mechanisms have traditionally been the basis for development of integrated approaches to reliability: the integrated approaches typically build around one or two isolated mechanisms.

- **Integrated Systems**: Several different groups have worked at providing reliability mechanisms in the framework of a general purpose operating system. Two such systems, the IBM System R [35] and the Tandem

Guardian Operating System [2] have integrated different sets of reliability mechanisms together with other operating system mechanisms to form an integrated operating system that provides a level of support for reliable transactions. Developing integrated systems is by far the hardest activity in the area of developing reliability mechanisms and this observation explains the sparsity of examples of integrated systems.

Much attention is given in the literature to the reliability problems and requirements of data base management systems. Two reasons are apparent for this emphasis: First, the basic importance of data capture and retrieval dictate that schemes for providing reliable operation be developed. Second, the nature of data base query and update activity is constrained enough so that the problems can be described and the nature of the interactions between components can be well understood. In some cases, solutions to specific data management problems can be generalized to apply to general operating system mechanisms.

4.5.2 Generic Approaches to Error Recovery

When a fault occurs, the impact on a system is to introduce an error into one or more of the data bases or state of the system. The purpose of error recovery is to eliminate the error from the system state. There are two generic approaches for performing error recovery: restoring the state of the system to a previously established (presumably correct) state, called *backward error recovery* and establishing a new state of the system derived from (the correct portion of) the system state containing the error, called _forward error recovery_ [1].

In backward error recovery (see Figure 6), when a fault occurs, the state of the system is reset to a previously established _recovery point_. Recovery points may take two forms: an initial value for the state of a system[3], or a value for the system state which was previously saved by some form of checkpoint operation. In terms of the figure, the horizontal line indicates system progress, the vertical bars are previous values of the state of the system, the asterisk indicates the occurrence of a fault. Implicit in the backward error recovery strategy is the assumption that subsequent progress after the recovery action has occurred will either avoid the same error (i.e., the error was transient) or that the path taken after the recovery will be different from the path that encountered the fault, and thus avoid the same fault. For the latter case to be true, the fault must be removed and this implies that a backward error recovery strategy must be coupled with a _reconfiguration strategy_ to eliminate the source of faults.

[3]in which case backward error recovery is just a system reset.

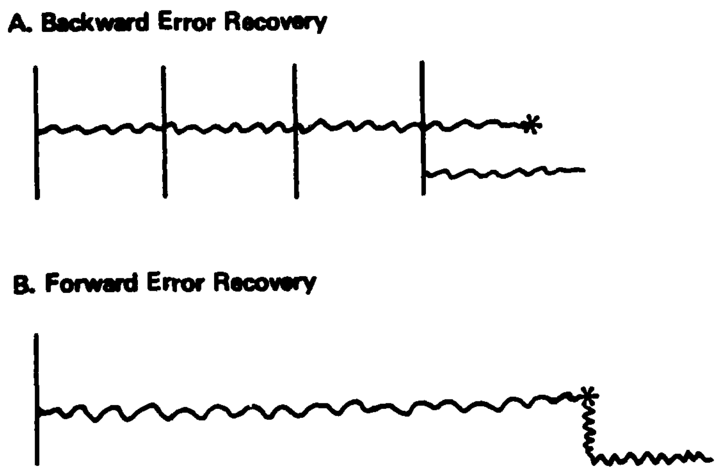


Figure 6. Backward vs. Forward Error Recovery

As the operation of a system progresses, checkpoints must be taken at intervals determined by the probability of a fault occurring, the cost of recomputing from a recovery point and the cost of storing intermediate system state. The decision to discard a recovery point is influenced by whether or not a fault will create a need for restoring the system state to that point. This is a major issue in distributed systems where an application may have several points of activity (e.g. requests to servers) outstanding at any one one instant.

In forward error recovery strategies, the application program takes a very active role in preparing for error recovery and in performing the actual transformations that correct an erroneous data base. Figure 6 illustrates the scheme of forward error recovery. A computation proceeds as the horizontal line until a fault (asterisk) occurs. At that point, a recovery action (vertical path) must occur which eliminates the effect of the fault and allows the computation to proceed, as if the fault had not occurred.

It is difficult to say anything in general about the recovery actions that occur in forward error recovery because they are highly tuned to the application in question. The term exception handling has been used to refer to the set of actions that occur when an error is detected in a forward error recovery scheme[4]. Exception handling is fairly well understood and a number of useful programming language abstractions have been developed [18], including one for the emerging DoD standard language Ada [27].

The main difference between backward and forward error recovery strategies is the amount of knowledge of the application program incorporated into the recovery strategy. With backward error recovery, the details of the application program are not important: the implementor of the application makes use of general purpose mechanisms for saving a recovery point, reconfiguring the application to eliminate the source of faults and causing the state of the application to be restored to a previously established recovery point. For forward error recovery strategies, much more of the mechanism to correct an erroneous data base has to be created anew for each application.

In terms of usefulness, powerful backward error recovery

[4]The reason the term exception handling has been associated with forward error recovery is that the scheme of going forward rather than backing up involves more explicit programming, and exception handling has been in the sphere of programming languages.

mechanisms have not been sufficiently developed so that forward error recovery strategies are not needed. While there is much ongoing work aimed at developing general purpose backward error recovery strategies and such strategies are more promising than forward error recovery schemes, it is still true that many existing instances of reliable application programs utilize forward error recovery strategies. Such applications are usually quite important, relatively simple and the reliability requirements are well defined. An example of one such application is the file system of an operating system. Preserving the integrity of the file system data structure is quite important and the nature of the data structure is relatively simple. With centralized hosts, the data structure of most file systems is a single centralized data base whose correctness is easily determined. Forward error recovery strategies are used extensively to recover the a data structure of a file system malformed due to errors introduced by a fault [60].

Finally, backward and forward error recovery strategies are not incompatible: it is possible to utilize both strategies in the same application in a complementary fashion [1]. In those cases where errors are simple and anticipated, forward error recovery schemes can be used. Backward error recovery strategies can be used in more complex cases where a fault has been unanticipated or to recover from a failing forward error recovery scheme. In fact, some schemes, such as the one proposed by Takagi [63], use both forward and backward error recovery strategies in the same general purpose error recovery mechanism.

In subsequent descriptions of specific mechanisms, the ideas behind forward and backward error recovery should be understood. The utility of proposed mechanisms can be judged based on the general utility of generic forward and backward error recovery schemes.

4.5.3 Generic Techniques Used in Specific Reliability Mechanisms

4.5.3.1 Introduction

In the course of the study, we looked at over 20 different reliability mechanisms -- specific schemes to aid in the reliable operation of application programs. There is a large degree of overlap as suggested by Figure 7. This Figure was derived from a diagram used in a talk by Saltzer in which he commented that a number of different mechanisms in operating systems were making use of several key ideas [56]. While each mechanism made a slightly different contribution to solving various operating system problems, there was a recurring set of general techniques

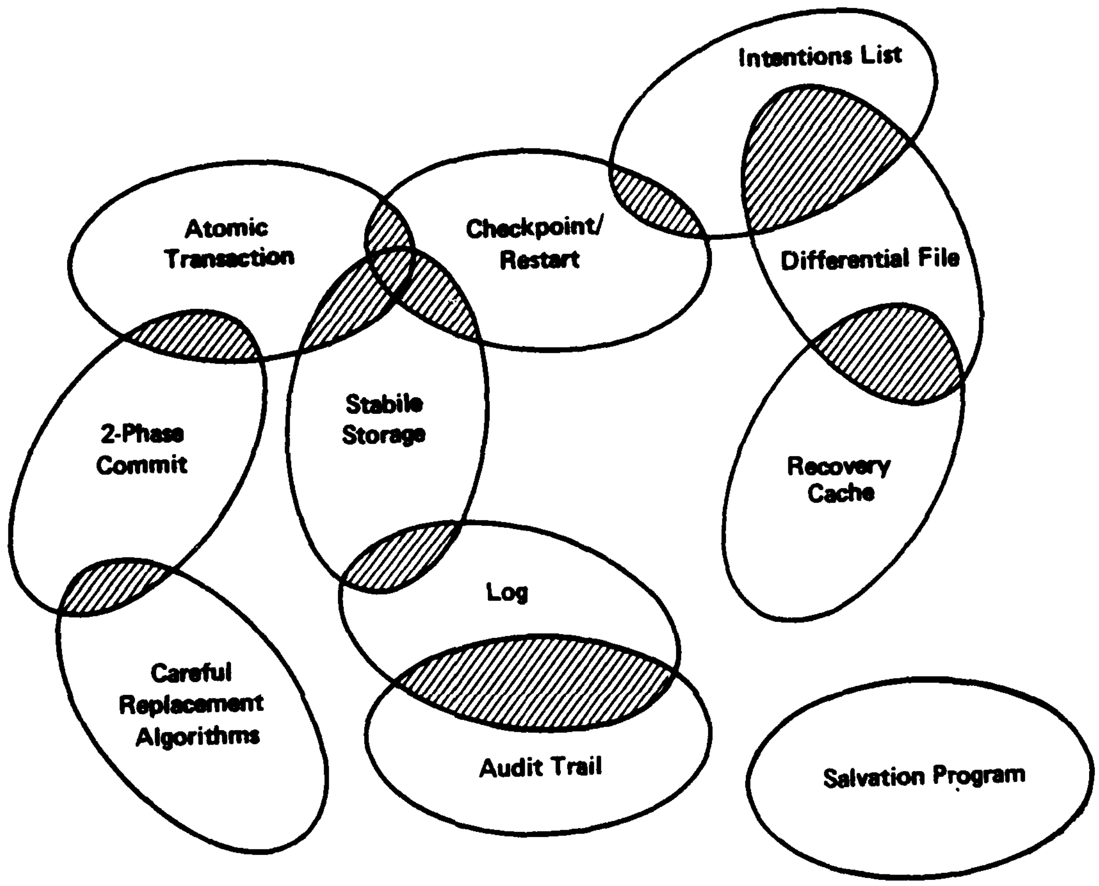


Figure 7. Recurring Use of General Techniques in Reliability Mechanisms

used in many of these contributions. In this section we will analyze some of the recurring or generic techniques used in the many different reliability mechanisms analyzed in this study.

4.5.3.2 Redundancy: Data and Control

Redundancy is at the heart of most reliability mechanisms. The reason for this is dictated by the effect caused by faults: a fault ultimately causes a loss of information or capability that is critical to the completion of a task. To make up for that loss, some form of redundant information or action must be utilized or invoked. Redundancy is most frequently viewed as a technique used to cover for losses of data, however redundant points of control are also frequently useful. An example of the use of redundant data is a distributed, replicated data base [64]. If a site holding one copy of the data base is unavailable for accepting queries or updates to the data base, an alternative site holding a redundant copy of the data base may be accessed. An example of the use of redundant points of control is the store and forward algorithms used by the ARPANet IMPs [20]. When a packet is sent by a sending IMP to a receiving IMP, knowledge of the packet is not discarded by the sending IMP until an acknowledgement is received. This knowledge represents a redundant point of control. If such an acknowledgement is not received before a timeout occurs, then an alternative receiving

IMP is selected and the packet is sent to the alternative.

Many different forms of redundancy are used in practice: duplicates, checksum codes, backup copies, and checkpoints are several techniques that are based on storing redundant information.

4.5.3.3 Atomicity

Atomicity is a technique that is being used increasingly in operating system mechanisms. Atomic operations are actions that occur as a complete unit. The undivided nature of atomic operations has several aspects:

- o The atomic operation will either occur completely, or will not occur at all. In particular, this must be true when an error occurs in the middle of an atomic action[5]. It is this aspect of the meaning of atomic action that is especially relevant to reliability mechanisms.
- o No other system activity will interfere with an atomic operation's use of resources. This aspect of the meaning is particularly relevant to mechanisms for controlling concurrent sharing of resources in a system by multiple processes.
- o The desired operations of an application program are programmed in terms of multiple sequential (or parallel) atomic actions.

---

[5]Talking about the middle of an atomic action is somewhat contradictory, however we assume that as with atoms in physics, our atomic actions are really composed of smaller parts that are packaged together into one atomic action.

From the standpoint of reliability mechanisms, atomic operations yield two desirable qualities: predictability of the effect of operations and bounding the period of time during which a fault can cause problems for an application program. Predictability in the face of failures is desirable because the extent of the error recovery operation is well defined. If a failure occurs and an operation is partially completed, partially changed data bases must be restored to their previous values. This can be a difficult task. Atomic operations are usually coupled with a mechanism for storing results in non-volatile storage. Thus when an atomic action completes, its effect is also made permanent. Such atomic operations limit the time during which a fault can cause a disruption. Once an atomic action has finished, the vulnerable period is over. This property of limiting the period of vulnerability to faults contributes to efforts aimed at putting lower bounds on overall system reliability.

4.5.3.4 Isolation of Partial Results

A number of different mechanisms attempt to keep intermediate or partial results separate from permanent data bases. The benefit of this strategy is that if, for some reason, the partial results of the computation need to be taken out of the permanent data base, this is easy to accomplish. In the case of reliability mechanisms, the reason is the occurrence of a fault in the computation. For other areas of operating system mechanism, reasons include: conflict between two or more processes over access to a resource, breaking of a deadlock, and eliminating the effect of an aborted, partially completed operation. Use of such techniques clearly enhance the ability to undo the effect of a computation and as a result are used as part of general purpose reliability strategies.

4.5.3.5 Permanence of Effect

Most reliability mechanisms incorporate some form of moving important data from volatile storage to non-volatile storage to achieve permanence of the effect of an operation. A characterizing distinction between volatile and non-volatile storage is the difference in probability that a device will fail to properly record a value. Volatile storage devices have a higher probability of failure than non-volatile storage media. Examples of non-volatile storage include magnetic and video disks and tapes, punched paper cards and tapes, printed paper copies and even possibly replicated copies of data stored on a volatile storage device.

Typically a reliability mechanism will insure that a data base is stored on a non-volatile storage media before making a commitment (e.g. sending a positive response to a request) to perform an action. Sometimes the old value of a record in a data base that is about to be updated is moved to non-volatile storage[6] so that a permanent record of the history of the data base can be kept.

4.5.3.6 Restoration of Acceptable State

One of the purposes of reliability mechanisms is to aid in the continued operation of a system. This means that the effect of errors introduced as a result of faults must be removed so that the normal assumptions about the syntax and structure of the data bases of a system are valid. There are several different approaches to this:

- o Restore a previously correct version of the data base and continue the operation of the system from this point. The effects of all operations that were performed by the system between the point where the previously correct version was stored and the fault occurred are lost.

- o Explicitly remove the error from the data base, in effect repairing the data base, so that it is in the same state it was in before the fault. It is possible that no information will be lost although this is generally more difficult to perform.

- o Force the data base into some acceptable state (possibly an initial state) from which normal system operation may

[6]typically called a journal or a log.

proceed. The forcing procedure typically examines the data base as a whole, keeping those records which follow the correct syntax of the data base and discarding records which do not fit. After this reconstruction procedure has taken place, the data base will be in some acceptable state, however there may be no single previous point corresponding to this state. This is quite a common form of data base restoration used in data management applications including file systems of computer systems.

4.5.3.7 Bounding the Time During Which a Fault Causes Problems

To avoid cleanup operations, several reliability mechanisms try to limit the time during which the occurrence of a fault will require extensive data base repair. Typically this is done by storing updates to a data base in a separate area and adding these updates to the main data base by a relatively simple and fast operation. For example, to add a group of elements to a list, the list structure of the group is built first. Then the group is added to the existing list by storing a pointer to the addition into the end of the existing list. If a fault occurs during the time when the additional list structure is being built, the existing list structure is unaffected. The time during which a fault can cause problems is limited to the time the pointer at the end of the existing list is being updated. This is a simple illustration of a general class of _careful replacement algorithms_ (See Section 4.5.4.8). A related technique is the set of _2-phase commit_ protocols for coordinating the application of a logically single update to multiple distributed data bases (see Section 4.5.4.9). The effect of such protocols is to put a window on the period of time during which a breakdown in communications will cause problems in coordinating updates.

4.5.3.8 Use of Timeouts

Timeouts are a pragmatic form of fault detection that are used extensively in distributed systems. One very practical judge of whether a task has completed correctly is to determine if it has completed at all. In systems where continued service is important, timeouts are used as the ultimate assessment of the health of providers of a service.

4.5.3.9 Summary

Specific reliability mechanisms make use of different combinations of the generic techniques (or approaches) described above. In the next section, a number of reliability mechanisms which have potential use in distributed system will be described. Each of these mechanisms applies the techniques described above in different ways, usually dictated by the nature of the problems being solved and the resources being utilized.

4.5.4 Specific Techniques

4.5.4.1 Introduction

Specific instances of reliability mechanisms require that variable aspects of generic approaches to reliability be fixed. In addition, even more specialization is required when an instance of a mechanism is actually implemented in a real computer system. In this section, we present a catalog of ten different specific mechanisms which are in current use in computer systems for providing reliable operation. For each technique, we consider the following two topics:

- o Description of general method
- o Example solutions that use method

Most reliability mechanisms have been developed for data base management applications and as a result have been oriented towards problems encountered in providing reliable access to data bases. Many application programs fall into the category of data base management, and as such can make use of standard mechanisms. Other applications do not fit this model, but still have requirements for reliable operation. General purpose mechanisms tend to be much less sophisticated in the nature of the operation performed by the mechanism. The focus for general purpose mechanisms has been to provide the basic building blocks which a programmer can use to program reliability into an application program. One area for future work is the development of general purpose reliability mechanisms which are integrated into the programming system (programming language, operating system calls) in which applications are being developed.

4.5.4.2 Checkpoint/Restart

Description of general method

Checkpoint/Restart mechanisms combine three of the generic reliability techniques described in section 4.5.3: data redundancy, permanence of effect and restoration of acceptable state. At intervals during a computation a complete record of the state of the computation (a checkpoint) is transferred to storage which has a high probability of keeping the state permanently. Magnetic tapes and disks are typical media used for checkpoint storage.

If a fault occurs subsequent to the checkpoint, then the only information that is lost is the state that was changed between the checkpoint and the point of the failure. Given that the failure was transient, a restart operation may be performed by restoring the state that was recorded at the checkpoint and continuing the computation. The rate at which checkpoints must be taken is influenced by several factors:

- The size of the state of an application.
- The probability of a fault occurring.
- The maximum permissible period of outage; the importance of timely completion of the application.
- The difficulty of taking a checkpoint at a given point in a computation.

Recording the complete state of an application program can be quite difficult, especially when the program is dealing with arbitrary devices whose characteristics are not well suited to being stopped or restored to a previous state. For example, it is quite difficult to reverse progress in a shared file system where multiple processes besides the one being checkpointed are making entries in catalogs. Thus, points in the progress at which an application program may be checkpointed are influenced by the nature of the activities occurring in the program.

Example of solutions that use method

As one of the most basic reliability techniques, checkpoint/restart mechanisms are found in a number of existing systems. The Guardian operating system [2] is a good example of the use of this technique in a distributed system.

4.5.4.3 Atomic Transaction

Description of general method

Atomic transactions attempt to provide a basis for order when the occurrence of a fault would normally create disorder. The generic techniques of atomicity, isolation of partial results, permanence of effect, restoration of an acceptable state and bounding the time during which a fault causes problems are used in combination to provide arbitrary user defined operations whose outcome in the presence of errors is well defined.

In the simplest form, an arbitrary number of application programmer defined operations (e.g., statements in a programming language) are bracketed by BeginAtomicAction and EndAtomicAction operating system calls. The atomic action mechanism guarantees that the group of statements bracketed will be executed as if they are a single instantaneous operation. At one instant their effect cannot be seen in the state of an application and at the next instant, either the entire effect will be observed or, if an error occurred in the execution of the atomic action, the state will not be altered. The instantaneous aspect of atomic actions, while useful, is not the primary attraction from the standpoint of reliability. Rather, the predictability of the effect of an atomic action is of primary interest.

Several techniques have to be combined to achieve the behavior described. First, changes in state that occur during an atomic action have to be reversible in case the occurrence of a fault prohibits completion of the action. This requires provision for reversing modifications that have been made to the state of the application program. For data base updates, reversal of modifications is usually done by keeping updates separate from the main data base and applying the effect of the atomic action to the data base as part of the processing of the EndAtomicAction primitive. For resource allocations, a corresponding deallocation must occur. Compensating for a failure in a resource deallocation may present greater difficulties because the same resource may not be reallocated due to interactions with other concurrent applications competing for the same resource.

There are a number of parallels in the requirements for backing up a computation between atomic actions and backtrack programming disciplines [17]. In problems suited to backtrack programming, there are usually a number of different alternative solutions that may be possible, but no knowledge of which one will be correct. The approach is to explore one possible solution until it is determined that it cannot be the solution and then to signal backup to the most recent branch point. All modifications to the data bases of the computation must be undone (typically by inverse operations) and then the computation resumed at the branch point. A maze solving program is an example of a problem well suited to backtrack programming. Techniques for reversing a computation developed for backtrack programming should be applicable to recovery operations for failed atomic actions.

Another addition frequently made to atomic actions is to cause the effect of the action to be made permanent upon termination. This way, once the EndAtomicAction has completed, it is certain that the effects of the action will not be reversed by a system crash. This primarily applies to data base updates and the meaning of permanent is that the updates are moved to some form of non-volatile storage (see discussion of stable storage below).

Atomic actions are attractive for many reasons. From the standpoint of reliability, they cause well defined behavior even in the presence of failures. It is this well defined behavior that permits an application program to take action which will compensate for the error introduced by a failure.

Example of solutions that use method

Numerous examples of proposed use of atomic actions exist [4, 9, 16, 36, 41, 50]. Perhaps the best discussion of a specific implementation of atomic actions is the description of distributed data bases and System R [35].

4.5.4.4 Log, Journal, Stable Storage, Audit Trail

Description of general method

A log, journal or audit trail is a running record of evolving modifications that are being made to the state of an application program. The purpose of such mechanisms is to carefully keep track of partially developed state information so that in the event of a failure, this information may be identified and manipulated (removed, reused, etc.). Logs combine several generic techniques of section 4.5.3 including isolation of partial results, permanence of effect and restoration of acceptable state. The model for logs comes from the auditing profession, where records of partially computed data are kept even after the final result is obtained so that if some question arises in the future it is possible to reconstruct the computation that lead to the final result.

Historically, logs have been implemented by writing to magnetic tape all pertinent information developed during the course of a computation. One reason tape has been used is because it is inexpensive and can hold the large amounts of data that result in such a recording operation. Current uses of logs do not view the log as a permanent record of evolving computations, but rather a temporary record that will be thrown away after the computation has successful completed. Thus, higher performance (and cost) storage devices such as disks can be used for logs. The type of information written to the log is usually data that cannot be reconstructed at some point in the future. Examples include, input typed by users of the application, the old value of a record about to be updated, and intermediate results on which further processing will be performed.

Consider the following example: The purpose of an application program is to keep track of geographical positions of military units. It does this by accepting input data from different sources, and updates position information based on this data. Since humans are involved in the typing of some input information, it is desirable to handle the information entered as carefully as possible. Also, since the information in the data base is critical, it is important to insure its integrity. A log could be used to record both backward and forward error recovery information.

A transaction (a period of interaction) is used as a bracket to determine when information should be stored in the log and when it is safe to discard the log; the period in which a human enters new position information and verifies the updated position is a transaction. Each new piece of information entered by the user is immediately placed in the log. In addition, once the military unit has been identified, the record of information about its old position is written into the log. If by chance, some part of the system fails before the new position record is added to the data base, two different types of recovery action can occur: First, a backward error recovery operation can be performed where the old value of the position record is reestablished[7] from the value stored in the log. Second, in a forward error recovery operation, the update can be tried again by using the human supplied input data stored in the log. If only partial information was recovered from the log, that much can be used and the user can asked to reenter a minimum amount of input data. After the record in question has successfully been

[7]This also requires that any partial remains of the new position record be removed from the data base.

updated, the entries in the log can be thrown away, thus ending the transaction.

Providing high performance logs which display a low probability of failure is non-trivial. Lampson and Sturgis [32] discuss the steps that must occur to build _stable storage_ out of ordinary memory pages and disk pages. Stable storage is an attempt to integrate logs into the normal style of accessing data from the program's address space. In effect, certain parts of the address space are identified as being capable of storing data more reliabily than other parts. Lampson and Sturgis's technique involves storing two copies of data in pages on the disk, comparing the two versions to insure their integrity, and a careful order for rewriting the two versions of a page. There are obvious performance penalties of this approach and suggestions have been made that stable storage can be built completely in hardware. Until performance penalties have been overcome, the amount of data which is required to reside in stable storage must be limited.

Example of solutions that use method

Like checkpoint/restart mechanisms, logs have been used to a great extent in data base management applications. Recent proposals by Montgomery [41], Takagi [63] and Lampson and

Sturgis [32] are good examples of the use of logging mechanisms. Again, System R is a good example of logs integrated into a complete system [35].

4.5.4.5 Intentions List, Differential File

Description of general method

The focus of intentions lists and differential files is on keeping partial results separate from a data base as a whole. These are high level concepts in the sense that both mechanisms are typically built out of lower level mechanisms such as stable storage or other logging mechanisms.

An intentions list is a dynamically growing list of data base changes that an application program would make if it were to complete successfully. A differential file is a data base whose changes are kept separate from the main data base. These two techniques are essentially the same, differing only in their emphasis. The rationale behind both approaches is that adding records to a large data base is an expensive operation which can be avoided by keeping all changes segregated in a separate file. (From the standpoint of reliability, differential files are useful because changes are kept separate from the unmodified data base). Accessing a record in the data base requires that first the change file and then, if no match is found, the main data base be searched for the record in question.

Example of solutions that use method

The best discussion of differential files is by Severance [57]. Lampson and Sturgis [32] describe a crash recovery mechanism which utilizes intentions lists.

4.5.4.6 Recovery Cache

Description of general method

Researchers at the University of Newcastle upon Tyne have developed a reliability mechanism based on a hardware device called a recursive cache [47, 24]. A recovery cache is a memory device in which previously established state information is held until the success of a computation involving that state has been ascertained. Application programs are expressed as a series of nested recovery blocks. As pieces of state information are altered by the statements of the programming language (e.g., assignment statements), copies of the old values of state are entered into the recovery cache. Each recovery block is associated with an acceptance test that is evaluated as the block is exited. If the acceptance test is passed, then the contents of the recursive cache is discarded. If, however, a fault has occurred during the execution of the recovery block, and the acceptance test is not passed, the pieces of state altered during the execution of the program in the recovery block are restored to the previous values stored in the cache.

In essence, the recursive cache is a fine grain log in which values may be easily entered and recalled. There are some questions as to the practicality of the recovery cache mechanism. First, the entire mechanism is based on the premise that if a recovery block fails the acceptance test, then there will be an alternative recovery block which can be invoked with the restored values and perform a functionally equivalent computation. Having to write the parts of a program in multiple independent ways seems to be questionable requirement. Second, the caching discipline seems to be indiscriminant. For the mechanism to work in a natural way, all values modified in a recovery block must be backed up in the cache. The load on the recovery cache for realistic computations would be extremely high -- possibly requiring twice as many write memory references as a program running without a recovery cache.

When an application is built out of multiple interacting processes the problems with recovery blocks grow. The problem is that a _domino effect_ can occur unless interactions between processes are carefully controlled [58]. A similar problem can occur with multiple interacting atomic actions.

The main benefit of the recovery block scheme is the demonstration of a mechanism in which hardware reliability devices and programming language constructs are completely integrated. Programs are written in terms of reliability requirements and considerations. This emphasis on integration is clearly a proper direction for future work in reliability mechanisms.

4.5.4.7 Salvation Program

Description of general method

A salvation program is used to force a data base back to a state where an application program can make further progress. The only guarantee made by a salvation program is that the state of the data base is one legal state -- not the most recent legal state nor even some previous state. A salvation program is typically invoked after some inconsistency in the data base has been detected. The reconstructed state is established by scanning through the structure of the data base, possibly deleting some uninterpretable data record and possibly constructing plausible linkages between unlinked records, based on a known property of records in the data base.

Salvation programs are frequently used to insure the integrity of file systems. After a system crash, the contents of the long term storage medium (typically disks) is examined to insure that the data conforms to a legal instance of the data structure of the file system. If it doesn't, then an attempt is made to reconstruct as much of the file system data structure as possible. This operation is greatly enhanced by self identifying data records (e.g. pages of a file), in which redundant information is kept about the position of a record in the larger file system data structure.

For salvation program techniques to be generally effective for application programs, they must be extended beyond the file system level and made available to individual application programs. The major advantage that salvation programs have over other reliability mechanisms is that they can make use of special knowledge about the correct structure of an application program's data structure. A file system salvation program incorporates knowledge of the correct form of data used to link together pages into files and files into directories. Great benefit and leverage in reconstructing a consistent data base results from such knowledge. A desirable, although infrequently provided extension to system salvation programs is to allow an application program a chance to examine and possibly reconstruct its data bases after a failure has occurred but before normal operation of the system resumed. A similar advantage could be exercised in forcing the application data bases back to a consistent state.

Example of solutions that use method

A good description of salvation programs is contained in a survey article by Verhofstad [69]. Two good examples of salvation programs are the Multics file system integrity mechanism [60] and the Alto disk salvager [33].

4.5.4.8 Careful Replacement Algorithms

Description of general method

The purpose of careful replacement algorithms is to avoid the possibility of partially updating a data structure by making the period when updates occur in place be as short as possible. This frequently is done by building an updated part of a data structure separate from the main data structure and then adding the updated portion by storing a single pointer to the new portion. Another approach is to turn flags on or off indicating that a record is or is not part of the main data structure. Whatever approach is taken for adding the updated portion to the data base, the operation must leave the data base vulnerable to failures for a short period of time.

Example of solutions that use method

Careful replacement algorithms are frequently used as part of other mechanisms. For example, the crash recovery algorithm

Lampson and Sturgis [32] employs a careful replacement
orithm to implement stable storage.

.4.9 Two Phase Commit

Description of general method

Splitting a modification to program state information into phases -- preparation and commitment -- is a rather general idea which appears in many mechanisms. The idea is to develop all data base modifications which require difficult operations in the preparation phase and then to cause those modifications to become part of the data base in the commitment phase. Careful replacement algorithms are a form of preparation followed by commitment.

With distributed computations possible confusion arises over the disposition of updates to parts of a distributed data base. eral independent researchers have developed protocols for ring that either all parts of a prepared update get applied data base or no parts get applied. Such protocols are quite lar and are characterized by the following quote from [35]:

> The two-phase commit protocol allows multiple sites to coordinate transaction commit in such a way that all cipating sites come to the same conclusion as to the disposition of the transaction. One site, the transaction coordinator, makes the final commit/abort decision after all the other sites in the transaction have already agreed to respect the coordinator's decision. During the first phase of the two-phase commit protocol, all sites of the transaction are queried as to

whether they can commit their part of the multi-site transaction (e.g. they haven't already unilaterally aborted). Each site becomes _recoverably_ prepared to go either way and awaits the coordinator's decision. Once all sites are prepared to commit, the coordinator makes its decision and all sites are notified to commit the transaction during phase two of the commit protocol.

Being prepared to commit means that the site will not require any additional resources to commit the transaction. Obtaining additional resources could cause deadlock ... . Once a site expresses its willingness to commit, it is no longer allowed to unilaterally abandon the transaction. Forbidding sites from unilaterally abandoning a transaction during commit processing compromises local site autonomy, but it allows the coordinating site to assume that sites will remain able to commit (or abort) until the coordinator is able to decide which way to go.

Example of solutions that use method

Several specific mechanisms have been used various forms of 2-phase commit protocols. Thomas [64] used an early form of this protocol to coordinate the application of updates to a distributed data base. Lampson and Sturgis [32] use a 2-phase commit protocol. A good description of alternative forms of 2-phase commit protocols appears in [35].

4.6 Integrated Approaches to Reliability in Distributed Systems

The major problem with the various mechanisms described in the previous section is that each mechanism is separate unto itself with no consideration of how it fits in with other mechanisms and services of an operating system. Providing basic reliability mechanisms that may be used by application programmers to program reliability is one step towards aiding the construction of reliable programs. A more helpful development, however, would be mechanisms that reduce the amount of application specific programming that has to occur to provide reliable operation.

What is needed is an integrated framework for programming applications that will lead naturally and easily to reliable operation. Several existing computer systems have started along this path, however we are still in the early stages of development of such systems.

4.6.1 Tandem Guardian Operating System

Tandem Computers has built an operating system aimed at supporting applications that have high reliability requirements. An example of one such application is automatic toll billing for telephone systems. Quoting from [29, 2], The Tandem/16 computer system is an attempt at providing a general purpose, multiple computer system which is at least one order of magnitude more reliable than conventional commercial offerings. Through software abstractions a multiple computer structure, desirable for failure tolerance, is transformed into something approaching a symmetric multiprocessor, desirable for programming ease.

The hardware of the Tandem system is typical of hardware intended for high reliability and availability: multiple redundant processors, power supplies, busses, disks, etc. The basic strategy employed in the software of the Tandem system (and the motivation of the name of the company) is a _buddy system_. From [2]:

> The system structure can be summarized as follows. Guardian is constructed of processes which communicate using messages. Fault tolerance is provided by duplication of components in both the hardware and the software. Access to I/O devices is provided by process pairs consisting of a primary process and a backup process. The primary process must checkpoint state information to the backup process so that the backup may take over on a failure. [The backup process normally executes a program which merely checkpoints data passed to it by the primary process. It is, however, capable of running the same program as the primary process in the event of the failure of the primary.] Requests to these devices are routed using the logical device name or number so that the request is always routed to the current primary process. The result is a set of primitives and protocols which allow recovery and continued processing in spite of bus, processor, I/O controller, or I/O device failures. Furthermore, these primitives provide access to all system resources from every process in the system.

Thus, the basic organizing principle behind the Tandem system is redundant control points which track the progress of an application program by recording its outputs to long term storage devices. The loose coupling of multiple components in the system makes dynamic reconfiguration relatively easy, since all interactions are by messages which are addressed to functionally named components. In the event of a failure, the backup process can pick up the computation by proceeding from the last checkpoint, utilizing data received in previous checkpoints.

The Tandem system provides the capability for developing reliable application programs. It is, however, the responsibility of the application programmer to program reliability measures into the program using the system provided primitives. Since these primitives are rather low level (as contrasted with BeginTransaction and EndTransaction primitives), considerable effort must be invested for each new application program developed on the Tandem system.

4.6.2 IBM System R

For a number of years, the IBM San Jose Research Laboratory has been investigating the design and implementation of relational data management systems in the form of System R. System R originally served as a vehicle for research in relational data management and user and programmer interfaces to relational data bases. With the advent of distributed systems, System R has been used as a testbed for new research in distributed systems and the problems of providing reliable and high performance access to a distributed relational data base.

The System R recovery facility [35] is aimed at solving reliability problems associated with:

- o <u>Transaction failures</u> (<u>aborts</u>): Undoing the effects of an incomplete transaction during normal system operation.
- o <u>Site failures</u>: Bringing the on-line data base to a consistent state following an unplanned interruption of service at a single site.
- o <u>Media failures</u>: Repairing damaged portions of an on-line, non-volatile data base.

Many different mechanisms are used in coordinated ways to achieve the desired results, including: atomic actions with a two-phase commit/abort protocol, forward recovery logs (logs that record actions that must be performed), backward recovery logs (logs that record previous state information), checkpoints which are used to limit the amount of log information which must be scanned during a restart operation and careful replacement algorithms for manipulating modified pages of the data base.

System R represents an engineering solution to providing a reliable distributed relational data management system. The techniques are used in a very well orchestrated manner, utilizing as much knowledge of the operation and accessing patterns of

System R as possible. The description [35] of how many different mechanisms dealing with reliability (as well as other system issues) indicates the delicate balance that must be created in order to end up with a system which displays significant improvement in reliability.

4.6.3 Other Integrated Systems

There are several other examples where reliability mechanisms have been integrated in with other mechanisms to produce one cohesive system.

The IBM Distributed Processing Programming Executive (DPPX) [30] is an operating system designed to support distributed processing with the IBM 8100 computer system. The Data Base and Transaction Management (DTMS) [70] portion of DPPX provides a reliable atomic transaction facility integrated with other system functions including data base management.

NASA has sponsored two different designs for ultra-reliable computing systems intended to control potentially unstable aircraft. Both the SIFT (Software Implemented Fault Tolerance) computer [71] and the FTMP (Fault Tolerant MultiProcessor) computer [23] make heavy use of redundant resources for insuring that the failure of any single point will not cause the failure of the system. In SIFT, "iterative tasks (the repetitive tasks that must go on in aircraft control) are redundantly executed, and the results of each iteration are voted upon before being used. Thus, any single failure in a processing unit or bus can be tolerated with triplication of tasks, and subsequent failures can be tolerated after reconfiguration" [71]. The FTMP design "is based on independent processor-cache memory modules and common memory modules which communicate via redundant serial busses. All information processing and transmission is conducted in triplicate so that local voters in each module can correct errors. Modules can be retired and/or reassigned in any configuration. Reconfiguration is carried out routinely from second to second to search for latent faults in the voting and reconfiguration elements. Job assignments are all made on a floating basis, so that any processor triad is eligible to execute any job step. The core software in the FTMP will handle all fault detection, diagnosis and recovery in such a way that application programs do not need to be involved." [23] A paper by Rennels [51] compares SIFT and FTMP. The special nature of these two systems (control applications) and the importance of the tasks on the safety of humans, justifies the rather expensive approach taken in both systems.

4.7 Summary of the New Reliability Requirements and Capabilities of Distributed Systems

Techniques for insuring the reliability of computer systems have been used for a long time -- probably from the start of the use of computers. Indeed, with the earliest computer systems that had a short mean time to failure, computations requiring longer than the mean time to failure had to have relatively frequent checkpoints taken, just to finish a normal run. Thus, the need for reliability mechanisms in distributed system is not new. Reasonable questions arise:

- What is new about distributed systems in their requirements on reliability mechanisms or in their capabilities to provide reliable service?
- *Do we need to develop new reliability mechanisms for distributed systems or are the mechanisms used in current single site systems sufficient?*

In this section we discuss three attributes that distinguish distributed systems from single site systems in the area of reliability: Independence of Failure, Need for Coordination and Need for Flexible Bindings. Each of these attributes require new approaches for providing reliable operation or modifications to existing reliability mechanisms.

4.7.1 Independence of Failure

The major new capability of a distributed system regarding reliability is that the parts of the system have the potential for operating with true independence. Emphasis is placed on potential, because it is also possible to build distributed systems in which the failure of one component will induce the failure of other components.

It has always been difficult to make convincing arguments about dynamic or real-time reliability mechanisms that insure continuous operation in a single site system. This is because the fault that caused the error will either cause an error in the error recovery programs or persist and cause another error if the system is somehow restarted. With distributed systems built out of many interacting components, a convincing argument can be made about the independence of the multiple components. Thus it is possible for one component to work perfectly even though another component has failed. The correctly operating component can perform error recovery operations to compensate for the effect of the failed component.

While many aspects of the isolation of one component from another are inherent in the architecture of distributed systems, some dependencies can creep into a system. Examples include:

o A distributed system built out of many components in a single building will not be able to tolerate the failure of electrical power to the entire building.

- A multi-component application program where one component acts as a central controller, will not be able to survive an error in the central component.

- A multi-component system in which the components are connected by a single communications path is susceptible to failures if the communication path is broken or flooded with messages from a malfunctioning component.

Thus, a distributed hardware architecture alone will not yield systems that are inherently more reliable than single site systems. The real issue is distribution versus centralization in all aspects of system operation: hardware, programs, data storage, communication paths, decision procedures, system monitoring, etc. The architecture of single site systems pushes most of these issues towards centralization. The architecture of distributed systems makes it possible to provide these features in a distributed way so that independent operation can be achieved. It is still possible, however, to end up with a system built on a distributed architecture in which many of the features are centralized with no independent recovery components and thus become a weak link from the standpoint of reliability[8].

---

[8]There may still be good reasons for building such centralized facilities on a distributed architecture. Performance characteristics might be one such reason.

The conclusion is that a major difference between distributed systems and single site centralized systems with respect to reliability is the potential for independent, isolated operation. This potential is just a foundation that must be carefully utilized to develop an entire system whose components are truly independent with respect to the propagation of errors.

4.7.2 Need for Coordination

Just as distribution admits new possibilities for robust operation, it also causes new problems due to the extra coordination required between the components of a distributed system. Reliability mechanisms that yielded effective insulation from failures in centralized systems either do not work, require added mechanism or exhibit poor performance when coordination is required between multiple components.

Some reliability techniques simply do not work (in their original form) in a practical distributed system. One example is salvation programs. Generally for a salvation program to operate properly, all activity modifying possibly corrupted data bases must cease. The salvation program views the entire data base and tries to make sense out of the remains of data that contains errors. For distributed systems, this would involve inspecting the entire distributed data base of the system and correcting inconsistencies that might possibly span component boundaries. This is both difficult and impractical in a distributed system; Difficult, because it is hard to stop multiple asynchronous systems all at the same point with respect to the salvation program. Impractical, because the nature of a distributed system is that unfailed components should be able to proceed as normally as possible without being impacted by the failures or recovery actions of other components. This runs counter to the philosophy of a salvation program approach to failure recovery.

One modification to the salvation programs which will work in a distributed system is to structure the data base of the system so that each component manages its own local part and allow a salvation program to be invoked on each part. The only coordination between the multiple components of the distributed system is the transmission of the order to start the salvation procedure. After that, each component proceeds to force consistency in its own private part of the data base. Notice that this approach is very similar to the use of atomic transactions and intentions lists in a distributed system. Distributed invocation of salvation programs and distributed application of intentions lists are representatives of a class of reliability mechanisms which require additional coordination mechanisms to deal with the multiple components of a distributed system.

The suggested modification of salvation programs presented above, suggests a general model for transporting a reliability mechanism from a single site system to a distributed system. The following scheme is a very high level view of the generic modifications that must be made to centralized mechanisms:

1. Start off with a mechanism that is invoked by the single control point in a centralized system and works on a logically centralized data base.

2. Design the distributed application program so that the previously logically centralized data base is distributed among the components. Each component has complete control over the records in its portion of the data base. Components cooperate in the distributed application program by exchanging messages and performing local data base queries and updates.

3. Where the centralized system formerly made a decision to invoke the reliability mechanism, design procedures for reaching a similar decision among the components of the distributed system. There are several alternatives here. One possibility is to have one distinguished component be responsible for initiating the recovery action. Another is to allow any component to initiate the recovery action, but to come up with a method of mediating conflicting recovery decisions. In any case, this coordination step can be quite complex and difficult to achieve, especially in the presence of errors.

4. Determining the scope of a recovery action among multiple asynchronous components is a difficult task. To make sense out of unsynchronized operations, the concept of a task or transaction is often introduced. In addition, all modifications to the data base performed during a given (uncompleted) transaction are labeled as such and in some cases, kept separate from

the main data base. Once the concept of a transaction whose effects on the data base are well distinguished has been developed, then communicating orders about recovery actions for a given transaction among the components becomes easy.

The coordination additions suggested above do have their price. There are generally two added problems: Performance and Complexity. A considerable amount of overhead is added to a centralized mechanism to make it function properly in a distributed environment. The use of transactions, keeping updates associated with a given transaction separate from the main data base, and the extra communication necessary to complete a transaction all decrease the throughput of a distributed application program. In addition, coordination procedures to insure that

1. Simultaneous independent decisions by separate components do not interfere with each other
2. Failures that may occur during the operation of a recovery procedure do not cause problems

add considerable complexity to mechanisms. Unless care is taken, the impact of coordination on the performance and complexity of a simple centralized algorithm can destroy its effectiveness.

4.7.3 Need for Flexible Bindings

The desire to have the components of a distributed application program bound to each other in a flexible manner arises from several different concerns. Reliability is one such concern. Others include a desire for scalable application programs and application programs whose composition of components changes over time. The common interest of each of these concerns is for dynamically changing bindings.

A binding is the means of attaching one component to another. In the context of distributed systems, the components are the separate parts of a distributed application each (possibly) executing on a separate single site system. Bindings take many different forms: communication paths, stored names, shared resourses (files, semaphores).

From the standpoint of reliability mechanisms, the interest is in flexible, reversible bindings between components. These are to be used in dynamically reconfiguring a system when a component has failed. For example, in the Satellite Image Gathering example of Section 3.3.2.2, multiple units are bound together to accomplish a task. If one of those units were to fail, we would like to get rid of all bindings to that failed component and make new bindings to some other spare component which could perform the same (or perhaps similar) function.

To motivate some of the issues associated with flexible bindings, let us consider a scheme in which the components of an application program are dynamically bound together by finding an instance of a generic component and using an identifier for the instance in primitives for sending and receiving messages. If a failure occurs in one component, then as part of a recovery operation, a new instance of the failed component will be created and the new instance will be brought up to date by replaying all of the interactions that occurred between the previous instance and the other components. At that point, interactions may resume with the new instance as if it were the old instance.

Figure 8 contains prototypical calls on six system primitives associated with flexible bindings. In this set of primitives, there are several different entities:

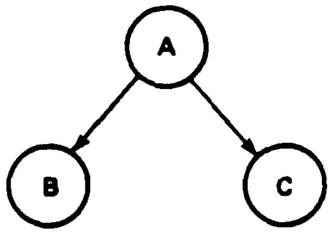
- **Instance:** An Instance of a component is one specific component chosen out of many components, all of which may provide a generically named service.
- **LogicalName:** The LogicalName is a character string representing the generic service.
- **Log:** A Log is storage used to record all of the interactions that occur between two components. Logs are used in the event of a failure to replay all of the commands issued by one component to a new instance of a generic component.

Figure 9 illustrates use of these primitives. In this example, a process A establishes bindings to processes B and C by use of the Bind primitive. In addition to establishing bindings,

```
Instance := Bind(LogicalName);
Log      := CreateLog();
Status   := Send(Instance, Request, Log);
Status   := Receive(Instance, Response, Log);
Instance := Rebind(LogicalName, Log);
Status   := DeleteLog(Log);
```

Figure 8. Operations on Flexible Bindings and Logs

Normal:

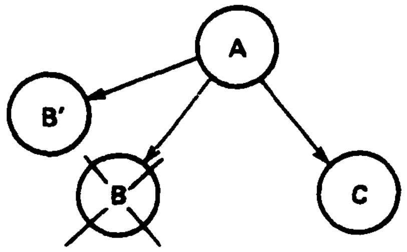


```
BInstance:=Bind ('B');
BLog:=CreateLog ( );
CInstance'=Bind ('C');
CLog:=CreateLog ( );
```

```
Send (BInstance,
      Request, BLog);

Receive (B Instance,
         Response, CLog);
```

B Malfunctions:



```
BInstance:=Rebind ('B', BLog);
```

Figure 9. Use of Flexible Binding Primitives

logs are also created for each set of pairwise communications. Normal operation occurs by process A performing computations, interacting with processes B and C by use of the Send and Receive primitives. Each time a Send or Receive is performed, the arguments are recorded in the Log associated with the interacting process. If no failures occur, then at the completion of the computation by A, the logs are discarded. If, however, an error occurs in, say process B, then a new instance B' is created using the Rebind primitive. In addition to the LogicalName, the Rebind primitive takes a Log as an argument. The contents of the Log are replayed to the new instance B'. For Send operations, the same arguments are sent to B'. For Receive operations, the results of Send operations by B' are examined to insure that they are the same results produced by B. Once the contents of the Log are exhausted, B' should be forced to the same state B was in just before the failure occurred in B.

This rather simple scenario gives a motivation of the possible use of flexible bindings. There are clearly some problems with the rather simple set of primitives presented above:

- o B' will not always be able to give an identical response as B to commands issued by A. Equality of responses must be generalized to be less rigid.
- o Pairwise interactions are not the only types of

interactions that occur in systems. A may send the results produced by B to C. B could also send requests sent by A to some third party D. High level knowledge about the structure of interactions between the dynamic components of a distributed system needs to be factored into the Request and Response logging mechanism.

This preliminary discussion of flexible bindings illustrates the important concepts, but just touches the surface of the mechanism needed. The true potential of independent components requires some mechanism for establishing and reestablishing flexible bindings.

4.8 Areas of Incompleteness

There are a number of areas where additional developments are needed before truly reliable distributed systems will be a reality. Such development are not in the area of new reliability techniques: existing modern approaches to providing reliability in single site computer systems appear to be adequate building blocks for reliable distributed systems. Rather, new developments are needed in integrating existing approaches into the primitives of real systems and providing the primitives necessary for coordination and configuration control in the distributed system. The following four topics need further attention in future research:

1. Integration of mechanisms into operating systems.
2. Appearance of reliability mechanisms in a programming system.
3. Coordination between components of a distributed system.
4. Use of flexible bindings in distributed systems.

4.8.1 Integration

Most of the reliability mechanisms discovered in the course of our investigation were stated apart from any considerations of how they fit in with the other mechanisms of an operating system or how they contribute to the reliability of the operating system itself. To be generally useful, reliability mechanisms have to be fully integrated with such mechanisms as resource control, interprocess communication and file storage mechanisms. In fact, each of those mechanisms will have a certain part devoted to providing reliable operation.

For example, assume that if a process A ceases to communicate with other interacting processes, the interprocess communication facility will time-out and notify a process B who has sent a message to A that a fault has occurred. To recover successfully from this fault, B must invoke aspects of the resource control and file storage mechanisms dealing with reliability to undo the effect of the failure. This would involve releasing resources acquired as a direct result of the interaction with A and undoing changes to files that were done in anticipation of completing the interaction with A. It is desirable for most of the actions related to the reliable operation of an application to be done by a general purpose mechanism.

Existing implementations of reliability mechanisms are typically built for special purpose applications such as data base management applications. New applications that do not fall into the data base management paradigm cannot use the reliability mechanism. Emerging systems such as the Tandem Guardian operating system do have reliability mechanisms built into the basic operations of the system and this trend should be expanded and encouraged in the future. There is a parallel here regarding protection mechanisms in operating systems. Early operating systems had minimal, if any, protection mechanisms built into the system from the start. As the requirement for protection in operating systems became apparent, protection mechanisms were <u>added</u> to the existing operating systems, with rather disappointing results. Information in the system was protected in many cases, but there were a number of lapses in the protection barrier. The lesson learned was that protection added as an afterthought does not create a secure system.

A similar effect is most likely to be experienced with systems where reliability mechanisms are added as an afterthought. Many areas of unreliable operation will be eliminated, however there will still be many unanticipated areas of weakness. This is why a system built from the start with reliability mechanisms integrated in with other facilities of the operating system is preferable over systems where reliability mechanisms have been added.

4.8.2 Appearance of Reliability Mechanisms in a Programming System

Mechanisms, alone, do not solve all issues associated with building application programs. A mechanism needs to have a representation in a programming system: language features, resource features, and operating system calls. Mechanisms need to be given first class treatment in programming systems in order to be used naturally and easily by programmers in applications. Many times the perceived benefit of a mechanism is judged by the understandable, easy to use representation of the mechanism rather than just the raw details of the service provided by the mechanism.

Relatively little in the way of language representation has been provided for reliability mechanisms. Language features such as exception handlers appear in languages such as PL/1 [26] and Ada [27]. Exceptions and exception handlers allow a programmer to invent names for abnormal situations in a program and write procedures for handling (correcting, cleaning up, aborting, etc.) these abnormal situations. The suggestion has been made that atomic actions can be specified by surrounding the body of an atomic action by operating system calls such as BeginAtomicAction() and EndAtomicAction(), and indicating atomic action failure by AbortAtomicAction(), however there seems to be more to the definition of an atomic action than this simple set of subroutine calls. For example, it might be necessary to specify the set of resources to be used during the execution of a transaction.

It is desirable to incorporate all of the semantics of a reliability mechanism in a programming language so that the compiler for the language can, at compile time, check to make sure the primitives are being used correctly and that proper optimizations are performed. In addition, a benefit of incorporating a syntax in a language for the facilities of a mechanism is that programmers tend to make better use of the mechanism since it is easy to use and well integrated with the other features of the language. For example, in the Pascal language, there are standard language definitions of files: opening, closing, reading records, writing records and positioning to specific records in a file. As a result, the use of structured data in programs and storing structured data on files is well supported and utilized naturally by Pascal language programmers. The Pascal compiler knows about the contents of files through precise declarations of the record structure of a file. Uses of the file primitives can be checked to insure that the contents of a record of a file are being read into a variable of the proper type. Many of the steps that a programmer in a language that did not have the concept of files would have to program explicitly are done automatically for a Pascal programmer by the file features in the language. In a similar manner, the meaning of reliability mechanisms can be incorporated into a programming language. Recent work by Shrivastava [59] has demonstrated one approach for embodying reliability mechanisms in a programming language.

4.8.3 Coordination between Components

Given the previously stated conclusion that distributed reliability mechanisms should be built out of coordinated use of reliability mechanisms at multiple single sites, there is a clear need for mechanisms for coordinating the actions of multiple components. An example of one such mechanism is the two-phase commit protocol [35].

There are many coordination (synchronization) mechanisms currently in use in tightly coupled multiprocess (and multiprocessor) systems. Examples include semaphores [10], monitors [22] and messages [5]. The problem with generalizing such mechanisms into a distributed environment is performance. The performance of the coordination mechanism has to match the granularity of the events being coordinated: if the events are small and frequent, the delay introduced by the coordination mechanism must be small.

Existing coordination mechanisms, such as the two-phase commit protocol tend to add a rather high overhead to the operations performed by the individual components of a distributed computation. As a result, the distributed reliability mechanisms that make use of the two-phase commit protocol must require infrequent synchronization to achieve good performance. For example, when a distributed atomic action is accomplished by using the two-phase commit protocol to coordinate multiple component atomic actions, the size of an atomic action must be relatively large to achieve reasonable performance. Lighter weight coordination mechanisms need to be investigated for those applications utilize reliability mechanisms that require more frequent coordination between components.

4.8.4 Flexible Bindings

The discussion in section 4.7.3 on flexible bindings presents a brief discussion of possible primitives for providing bindings between components that may be established dynamically and may also be broken and reestablished in a manner that allows a distributed application to proceed in the presence of failures. That discussion only touches the surface of this topic.

Questions that need to be answered include:

- What is the best form and appearance of reversible bindings?
- What data structures are needed to support flexible bindings? What type of data must be transferred among the components of a distributed application?
- What are the primitive operations that may be performed on bindings?
- How do spare components to which bindings have been newly established receive all of the state information that a broken component had developed?
- What is the best instantiation of flexible bindings in a programming language?

Research needs to be done in at least the areas outlined in this section before reliable, distributed application programs can be written in an easy and natural way.

5. GLOBAL RESOURCE MANAGEMENT

5.1 Introduction

A computer system needs the "raw materials" of computing in the form of hardware, input data, and programs in the same way that a business needs workers, raw materials, and tools to fashion its products. A business with plants situated in different cities must devote some effort to planning the movement of people and goods in order to reduce costs. Many factors are involved in this planning, from major long-term decisions such as where to locate a new plant, to the day-to-day operational decisions about feed stocks, inventories, and the size of the work force. Generally, higher levels of management are responsible for long range planning while lower levels handle more immediate decisions. External constraints (e.g., the location of a coal field) force some decisions but in other cases there may be great latitude (e.g., assigning production of a new product to one of several factories).

When computer systems become distributed similar effects are observed. This chapter concerns the nature and scope of the planning decision necessary for the efficient operation of a distributed computer system; we call this _global resource_ management. The focus is on the role of global resource management in meeting performance goals. The alternative aspects of the proof of correctness of the resource management strategies and reliable resource management will not be addressed.

Administrators may treat any computer system, distributed or centralized, as a "black box" and dictate the resource management policies to be obeyed by the box without regard for its internal structure. Thus it should not be surprising that distributed and centralized systems can share the same terminology for policy statements about resource management.

Even for centralized systems in which the resource management problems are simpler[9] resource management is rarely treated comprehensively or formally. Management policies tend to be implicit in system code, and not well documented or understood, for a variety of reasons:

1. Different resource managers are often designed by different people, resulting in widely divergent management strategies.

2. As systems are "tuned" over time, tests for special cases tend to accumulate in a resource manager with the

[9]because there are fewer resources and processes, and because the system topology is simpler.

result that the total behavior of the manager is undocumented or undocumentable, and the interactions of the special cases become incomprehensible even to the system programmers.

3. Management policies are often chosen with hidden goals. For example, priority may be given to one class of jobs provided that the processor utilization is not depressed "too far" -- and the limit is never quantified.

Before proceeding to the more difficult problems of global resource management, therefore, we take special care to clarify the language and goals of resource management in general.

In distributed systems the resource management problems are not so much _different_ from centralized systems, but rather enormously _magnified_. There are many more processes and resources, many more management decisions per unit time, and very complex system interconnection topologies. All of these factors exacerbate the resource management problems in distributed systems.

There are three major limiting factors to the benefits that can be achieved by global resource management:

1. Message delays between components of the distributed system.

2. The extent of administrative authority across geographically dispersed sites.

3. The complexity of the resource usage patterns of the system workload.

By making the message delays small enough, and administrative controls strong enough, a distributed system can be made to appear essentially identical to a single site multiprocessor with shared memory. Without some administrative control over the resources at different sites, global resource management is impossible since local resource managers could arbitrarily withhold resources from distributed tasks. Thus we assume in the remainder of this section that message delays are substantially longer than in single site systems, and administrative control over at least some distributed resources is possible.

The sheer size of distributed systems as measured by the number of processes and resources which interact within them is the greatest impediment to effective resource management. Operating systems theory has produced optimal algorithms or near-optimal heuristics for only the simplest resource usage patterns, often under unrealistic assumptions. For example, a large body of theory is concerned with deterministic processor scheduling; almost all of the results depend upon prior knowledge of the resource demands of the executing processes, and consider requests for only a single resource. These are both unacceptable assumptions in a real system. In addition, deterministic scheduling theory does contribute cause for pessimism, since in many cases it can be shown that optimal scheduling problems become computationally difficult (NP complete) when the number of interacting processes exceeds two or three. We begin with the understanding that global resource management will not be based on optimal scheduling.

The remaining paragraphs in this section define terminology and provide examples to clarify the fundamental concepts. Section 5.3 discusses the importance of the amount of time available to plan and execute plans for resource management. Section 5.4 describes the elements involved in formulating policies for resource management. Section 5.5 turns to resource management problems specifically associated with distributed systems, and classifies the major design decisions. Section 5.6 discusses two global resource management strategies in depth, stating policies and defining the mechanisms necessary to achieve them.

5.2 Terminology and Fundamental Concepts

5.2.1 Processes and Resources

At the heart of resource management are the concepts of processes and resources.

The process concept is often employed but rarely formally defined[10]. For our purposes we adopt an informal approach, and define a process as a time ordered sequence of *states*, where the transitions from one state to another are made by an active agent called a *processor* following the transition rules or instructions of a *program*. Thus, processes imply the existence of a processor and a storage medium to hold representations of the state and program. Processes accomplish all of the useful work of computers through their state transitions. Modern operating systems give the user the means to create new processes, organize processes into hierarchies, transmit messages between processes,

destroy processes. Even so the typical user of a single host sy em has one associated process (executing a program such as an editor, compiler or mail program) which is active at any given time. The user of a distributed system, on the other hand, must have at least one process active at each site involved in a computation. Our experience with the National Software Works suggests that many more processes will probably be involved.

A *resource* is an object that a process must possess in order to make computational headway. Resources may be indivisible

---

[10]A formal definition has recently been described in [39].

(e.g., a lineprinter can only be used by one process at a time) or divisible (e.g., memory can be partitioned among several processes). A process may need a threshold amount of a divisible resource to proceed (e.g., a tape merge requires at least 3 tape drives) or the rate of progress of the process may be an increasing function of the amount of a resource it possesses (e.g., a process receiving 10% of the processor proceeds faster than a process receiving 5%). A resource is capable of being possessed by at least two processes (otherwise there is no need for management), and observes an exclusion constraint which prohibits it from being owned by all processes at all times. In order to acquire a resource it does not possess, a process issues a _request_ for the resource; to designate that it no longer needs a resource, a process _releases_ the resource. _Request_ and _release_ operations may be explicit actions of a process, or implicit in some aspect of its behavior.

In most general purpose computing systems both primary memory and the central processor are resources, because a process cannot proceed unless it has both. Other examples of resources in these systems are disk volumes, disk controllers, tape drives, and communication lines. In any given system the set of components which are regarded as resources depends upon the system's structure. Real-time systems, for example, dedicate primary memory space to high priority processes to minimize process switching time. Because these processes cannot be delayed by requests for primary memory, for them, primary memory is not a resource.

All of the resources mentioned above are _physical resources_, pieces of machinery that are distinguishable in the computer room. Through software, _virtual resources_ can be created in a manner analogous to the implementation of virtual memory from physical memory. Processes can _request_, use, and _release_ virtual resources just as they do physical resources. Perhaps the most prominent example of virtual resources is a database lock. A process is required to acquire the lock before accessing the database or some part of it, and to free it when it is through.

Resources exist at different levels within a computer system and change their appearance from different viewpoints. Typically there are resources visible to the processes of the operating system which are invisible to user processes (e.g., process control blocks and I/O buffers). Resources at one level may be utilized to implement virtual resources visible at a higher level, following hierarchical design principles. For example, individual disk blocks, volumes, and controllers are rarely visible as resources at the application program level, having been recast into the more abstract virtual resources, files and data records.

5.2.2 Resource Management

Since all work done by a computer system is done by processes, the execution of a _request_ or _release_ instruction must be carried out by a process. We call such a process a _resource manager_. For most of this discussion we assume that each type of resource is managed by a separate resource manager[11], and we will view the _request_ and _release_ instructions as interprocess messages to the resource manager from its _clients_, the processes which use the resource. The resource manager always possesses the idle units of its resource, i.e., those not possessed by its clients.

A distributed computation by definition includes processes on several hosts. _Global Resource Management_ refers to the use of global information about distributed tasks to control the

[11]This is an important assumption. Briefly, independent managers for resources increase the possibility of deadlock and decrease resource utilization; monolithic managers for several resources are extremely complex to build and difficult to document.

physical resources on individual hosts as well as virtual resources that are composed from more primitive elements, possibly residing on several hosts.

A resource manager has responsibilities in two areas: logically correct behavior and performance[12]. A resource manager behaves correctly if it obeys the exclusion constraints for its resource, and responds appropriately to _request_ and _release_ messages; we do not pursue correctness further. Because a manager can choose to arbitrarily delay any process issuing a _request_, and perhaps can preempt resources or reclaim them immediately from unwilling processes, the manager can control the rate of progress of its clients to some degree. This control does not affect correct functioning but manifests itself in system performance measures (response time, throughput, etc.).

The extent to which a resource manager can affect performance depends critically on the information the manager can obtain about the future pattern of _request_ and _release_ messages. With no information available other than the _request_ and _release_

[12]Managers are also occasionally concerned with security, i.e., the prevention of unauthorized access to resources, but this is not relevant here.

messages as they occur, a manager can do nothing more than to maintain correct function.

Managers can obtain additional information that can influence allocation decisions, of four basic types:

$$\left.\begin{array}{l}\text{1. imposed}\\ \text{2. deduced}\end{array}\right\}\quad\text{absolute}$$

$$\left.\begin{array}{l}\text{3. statistical}\\ \text{4. volunteered}\end{array}\right\}\quad\text{probabilistic}$$

The first two types are utterly dependable; barring system failures, the information is guaranteed to be correct. The latter two types are not completely dependable, and must not be used to make decisions which could affect the correctness of the manager.

Imposed facts are known to be true because the resource manager itself imposes them. For example, a processor resource manager might impose a 10% duty cycle on a process, and therefore knows that over some time interval the process will consume 10% or less of the processor resource. _Deduced_ facts are obtained by analyzing some (unchanging) system components outside of the resource manager. For example, the resource manager might inspect the code sequences of a client process to deduce its future resource needs. This would only be possible if the process were prevented from changing its own program and thus invalidating the deductions. Statistical information as the name implies is derived from a series of observations of some quantity. The statistical analysis may be done offline and the results built into a resource manager as statically known facts. Alternatively the manager may make observations and attempt to follow dynamic trends. However the information is garnered, it is not completely trustworthy. Statistical information is often stated in terms of distributions, e.g., working set size distributions. Volunteered facts are given to the resource managers directly by clients through some form of interprocess messages. A client might state its intention to use a quantity of a resource over a period of time, or not to use the resource. The utility of this information depends on the trustworthiness of the client process. If a client volunteers information which is incorrect then decisions made by the manager based on incorrect information may lead to worsened performance.

5.2.3 Policies and Mechanisms

A policy is a goal or guideline set by the system administrator constraining the decisions made by a resource manager. An intelligently formulated policy is based on the value of the work performed by client processes, in an effort to maximize an overall benefit measure for the system. Insofar as policies hinge on "value received" they necessarily concern goals and priorities external to the computer system. For example, a system-wide policy might dedicate late night hours to one project known to absorb large amounts of computing; or it might grant preferred status to interactive processes over batch processes because it is felt that people should not be kept waiting.

A _mechanism_ is an internal system structure designed to implement a class of policies. An interesting discussion of mechanism versus policy can be found in [34]. The distinction between mechanism and policy is difficult to state precisely, and is best presented through an example, which follows.

Single host operating systems sometimes divide the processor manager into two components, the _dispatcher_ and the _scheduler_ or _policy module_. The dispatcher maintains a list of processes requesting the processor, sorted on a numeric priority field. When the currently active process releases the processor or exhausts its time quantum, the dispatcher activates the process with highest priority on the list. The dispatcher implements a mechanism for priority scheduling but does not determine a policy, because priorities are set by the scheduler. The scheduler executes asynchronously with the dispatcher and from time to time changes the priorities of the other processes. The scheduler decides the relative value of client processes, and assigns priorities accordingly. The scheduler implements a policy which is carried out by the dispatcher; the dispatcher could be mated with different schedulers at different times to implement a range of policies.

Below we use the term _strategy_ informally to mean a specific policy combined with an implementation mechanism. Ultimately we would like to make quantitative statements about the performance properties of various strategies, but in this report we do not progress so far.

5.3 The Importance of Time

Resource management strategies are realized as computer programs, and computers do not function instantaneously. Inevitably, delays in the decision procedure cause transients in system metrics when viewed on a fine enough time scale. This section attempts to clarify the role of time in resource management. We ignore for the moment the problem of maintaining synchronized clocks in a distributed system, and concentrate on the interplay of event rates and resource management decisions.

From one perspective we see a resource manager as a simple

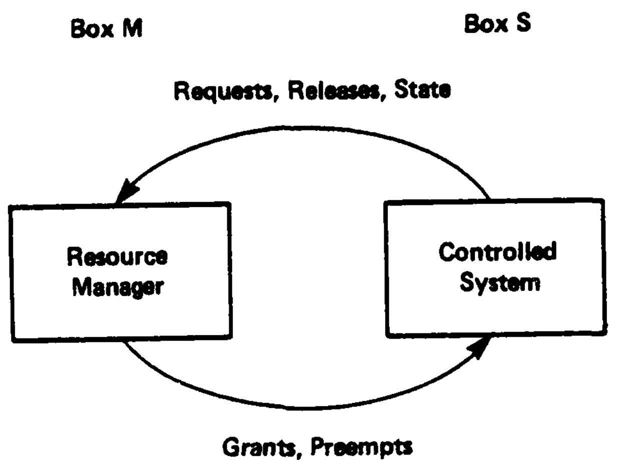


Figure 10. A Control Model of Resource Management

control system (Figure 10). In this model the computing system is partitioned into two boxes, one containing the resource manager and the other containing all remaining system activities. [illegible] es, Box M observes state changes in Box S -- Request [illegible] messages, the number of active clients, consumption [illegible] other resources that correlate with the consumption [illegible] resource, etc. Box M transmits control signals [illegible] in a effort to minimize an error function, E(S), on the state variables of Box S. At any instant in time, the absolute magnitude of E(S) is a measure of the deviation of the state of Box S from its "desired" or "optimal" state. The control signals are of two types: grant messages, the reply to a request indicating resource allocation and permitting a process to proceed, and preempt messages, forcing a client to free a resource and thus become blocked.

The control loop in this model consists of three parts: information gathering, decision making, and execution. In any practical system each of these parts takes time, and the minimum response time of the control loop to a change in the state of Box S cannot be less than the sum of the minimum times for each part. In a distributed resource management system, it is likely that the information gathering and control paths are by means of network message transmission. In this case the maximum network round-trip message delay limits the response rate of the control loops.

	Time Interval	Activity
manual techniques possible	>24 hours	physical reconfiguration
	about 1 day	daily operating procedures
automatic techniques necessary	1 to 8 hours	user session
	1 to 60 minutes	program execution
	0.1 to 5 secs.	user interaction
	about 50 msecs.	process-to-process interaction

TABLE 1. Time Epochs for Resource Management

Any resource management mechanism is effective only when evaluated over a sufficiently long period of time; the approximate minimal interval for success we call the grain of the mechanism. We can characterize the feasible resource management mechanisms on the basis of grain size. The classification into six groups shown in Table 1 is not absolute nor are the boundaries between categories sharp, but it illustrates our understanding of the pragmatics of resource management. The six grain sizes are discussed in turn below.

5.3.1 Grain Size: Days, Weeks, Months

Over very long time periods significant human planning and negotiation is possible. A time scale of weeks or months permits databases to be moved, new hardware acquired, and new software written to meet long term resource needs. Capacity planning is a routine commercial practice for large computer systems, and the resource management strategies at this level are usually considered in the province of Operations Research.

Optimization techniques based on linear or non-linear models may be used to aid the decision process. Costs are expressed directly in dollars since the planning period is of the same order as budget cycles. Capacity planning is closely related to the goals and financial status of the hosting organization.

5.3.2 Grain Size: One Day

Over periods of time approaching one day, human control of resource management is still possible but will probably be stylized. For example, computer operators may alter system configuration throughout a 24-hour day to track daily trends, following procedures in an operator's handbook. Management strategies are planned in advance to meet contingencies.

5.3.3 Grain Size: Several Hours

A typical interactive _user session_ lasts at most a few hours. A session corresponds to the period of time a user is in continuous contact with the system, usually beginning with connection and authentication (login) and ending at a user command (logout). Because session initiations occur very frequently resource management decisions made at this and smaller grains are automated[13].

5.3.4 Grain Size: Several Minutes

The next smaller grain of management is the _job step_. The growth of "command language" or "shell" programming has made it

---

[13] A common exception is the allocation of tape drives, since the tape handling sequence is difficult to automate.

difficult to define what a job step is, but to us it represents a significant computational task such as compiling a program, editing a file, or performing a statistical analysis of a dataset. Job steps can be separated into two types, interactive and non-interactive. Interactive job steps include editors and mail programs, each job step composed of many interactions between a user and a process. Non-interactive job steps include compiling and batched data analysis. Interactive job steps may persist for many minutes while the user issues individual commands; non-interactive job steps are usually brief, less than 5 minutes if the user must wait for results.

5.3.5 Grain Size: One Second

The basic command response interval should be on the order of one second for the most common user interactions. Delays longer than about two seconds interrupt the user's actions sufficiently to prevent continuous type in. When a user depends on hand-eye coordination to perform a task such as moving a cursor to an object on a screen, the response time must be even smaller.

Timesharing systems have adapted to this basic interaction rate. The scheduler of the TENEX operating system, for example, contains several time constants that are related to the interaction event rate, and on this system it is difficult to activate processes at a higher rate. The time constants control the frequency of scheduler runs, the frequency certain queues are inspected for wakeup events, the quantum size, etc.

5.3.6 Grain Size: Ten Milliseconds

At this level both the events and resource management mechanisms must be automated. High rate interactions arise between separate computer systems, between computer systems and sensors, and between computer systems and controlled mechanical devices. One very important case of this type is process-to-process communication. In the ARPANET, for example, message switching processors continuously allocate and release buffers on the basis of messages received from other processors; buffer _request_ and _release_ operations occur at intervals of 10 to 100 milliseconds.

5.3.7 Policies and Time

A resource management policy is an assertion about resource utilization by processes over time. Intuitively we recognize that policies are implemented by computer programs which take time to reach decisions. Consequently, over an interval on the same order as the grain of the resource management mechanism it is not possible to achieve meaningful control over resource allocation. A feasible policy must take this into account by making assertions on system measures averaged over an interval larger than the grain size.

Informal statements of policy rarely include the time constants involved. A few typical informal policy statements are:

1. "All processes should be treated equally."
2. "A process should receive a fraction of the processor proportional to its pie slice."
3. "High priority jobs should run first."
4. "Always run the process with the smallest product of size times processor time used."

The most direct example of grain is provided by the processor resource of a single processor. A group of processes may request the processor simultaneously but only one may use it, so that for very short intervals the processor utilization of one process is 100%, and for all others it is zero. The policy (1) above cannot be implemented over intervals as short as this.

The next paragraphs give an example of a formal policy statement. The policy is a minor variant of the TENEX pie-slice processor scheduling algorithm, chosen because there are concise informal and formal policy statements, and because the significance of the policy parameters is well understood. The policy was originally devised for a single host system; we will discuss the possibility of a distributed implementation in Section 5.6. The policy statement itself is independent of the distributed or non-distributed nature of the implementation.

5.3.7.1 An Informal Definition of the Pie-Slice Policy

The pie-slice processor manager attempts to divide the available processor time among a set of processes in proportion to administratively assigned pie-slices. A process P has a pie-slice S_P assigned to it, such that the sum of the S_P over all P is one. At any given time a subset of the processes are <u>demanding</u>, that is, would utilize the processor if it were granted to them. The processes which are not demanding are in wait states, for example waiting for disk I/O to complete.

A more careful statement of the pie-slice policy must take into account that only one process can use a processor at any instant, and that a process which does not demand its fair share over the long term cannot possibly acquire it. Because there are often large blocks of unutilized processor time most processes will not generate a demand equal to their pie-slice over a long period of time. As a result, the informal statement will be seen to hold only over an interval long compared to the process switching time, but short compared to a day.

5.3.7.2 Moving Time Averages

An important vehicle for making policy statements is the moving time average. Let $U_P(t)$ be the utilization of the resource by process P at time t. For the processor resource $U_P(t)$ is always either 0 or 1. The moving time average of the utilization is defined to be:

$$A_{P,r}(t) = \int_{-inf}^{t} U_P(x) r e^{-r(t-x)} dx$$

where r is a time constant determining the weighting of recent and less recent utilization values. r is sometimes chosen from the relation:

$$r = -\frac{1}{t_h} \ln(1/2)$$

The value t_h is the time interval required for the contribution of an impulse in $U_P(t)$ to decay to 1/2 of its original magnitude. The moving time average represents a history-sensitive approximation to the instantaneous utilization of a resource. If the time constant t_h is chosen to be large relative to the rate of change of $U_P(t)$, the effect will be to smooth the utilization values; as t_h goes to zero, $A_{P,t}$ converges to $U_P(t)$.

5.3.7.3 A precise Statement of the Pie-Slice Policy

The pie-slice policy can now be defined in terms of two time constants, t_1 and t_2, and a time interval t_i.

For all processes P and at all times t:

1) $A_{P,t_2}(t) < S_P$

2) if P has been demanding since $t-t_i$, $A_{P,t_1}(t) > S_P$

Clause (1) states that over time periods of about t_2 units or more, the amount of the resource consumed by P does not exceed its pie-slice. Clause (2) states that over short periods of time, if P is demanding then P receives at least its fair share. Together they imply that over moderate periods of time P receives the portion of its pie-slice which it can effectively use.

This example illustrates important features of policy formulations:

1. A precisely stated policy must involve a grain size, a minimum time interval over which it is effective.
2. Policies tend to have several components describing behavior over different time periods or load conditions.
3. Policies with simple intuitive bases may have formal specifications which are difficult to understand.

5.3.8 A Design Principle

The grain of a resource management mechanism is a fundamental limit to the range of policies that can be enforced by the mechanism. An important principle of system design is that the grain must be smaller than the mean interval between allocate/deallocate events to assure stable operation. This suggests that a feasibility evaluation be performed for each resource management mechanism during the design process.

The evaluation can be done in three steps:

1. **Determine the Grain Size.** Approximate the time required for a request or release to be serviced, by adding known message delays and the manager's processor requirements.

2. **Determine the Allocate Event Rate.** Estimate or measure these characteristics of the system load.

3. **Compare the Grain and Event Rate.** If the mean interevent time is smaller than the grain, the mechanism is infeasible.

Even when only rough approximations to the rates and delays can be obtained, the procedure above may yield valuable information about the eventual performance of the resource manager.

5.3.9 Example

Consider performing dynamic task assignment to the APU's described in Section 3.3. Assume an FCU configured with five APU's, a data collection module, and an archival storage module as shown in Figure 11. The data collection module buffers sensor

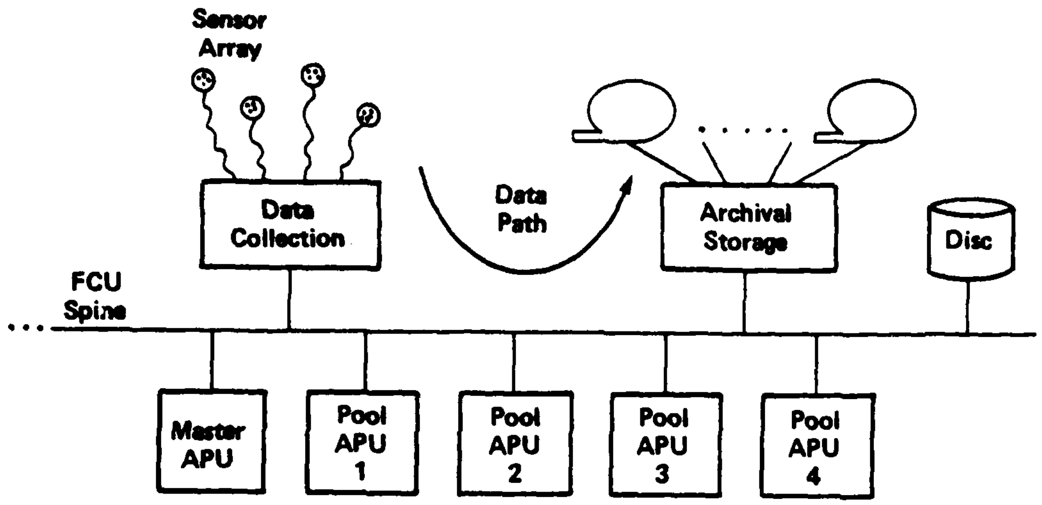


Figure 11. A Possible FCU Configuration

data internally, and at intervals of about 0.5 seconds must transfer a buffer to the archival storage unit. The transfer is done under the control of an APU which moves the buffer from the data collection module, into its internal memory, and then to the archival storage unit.

Suppose two designs have been proposed for the APU resource management. Design A would dedicate one APU permanently to the transfer task; design B would require the Master APU to select one of the Pool APU's dynamically for each buffer transfer. A dynamically assigned APU must obtain a copy of the task image from disk storage before it can initiate the task. The message sequence for Design A might be:

1. The data collection module sends a "buffer ready" message to the dedicated APU (say, Pool APU 3).
2. APU 3 responds with an acknowledgement and permission to transmit the buffer.
3. The data collection module transmits the buffer to the APU 3 local memory.
4. APU 3 transmits the buffer to archival storage, and waits for an acknowledgement.
5. Archival storage acknowledges with a "buffer accepted" message.

The message sequence for Design B might be:

1. The data collection module sends a "buffer ready" message to the Master APU.
2. The Master APU selects a Pool APU (say, Pool APU 2) to service the request, and transmits a "task assignment" message to APU 2.
3. APU 2 receives the "task assignment" message and transmits a "load task image" request to the disk.
4. The disk transmits the task image to APU 2, and APU 2 initiates the task.
5. APU 2 proceeds from step 2 in the previous sequence to complete the transfer.

We follow the steps above to determine the feasibility of Design B. The time for a <u>request</u> from a data collection module to be serviced is the time required for the Master APU to select a Pool APU, transmit the "task assignment" message to it, plus the time required for the selected APU to load the task image from the disk. Suppose small messages can be transmitted between FCU modules in 10 milliseconds (including low-level software overhead), and the task image can be transmitted in 20 milliseconds. The Master APU will process the request in 5 milliseconds or less. Thus the elapsed time for a request is approximately 3*10+20+5 = 55 milliseconds, assuming no delay in obtaining a free APU. The allocate event rate as stated above is 2 buffers/second, well below the maximum serviceable request rate. Therefore Design B is feasible.

If the allocate event rate were 20 buffers/second, the interevent time of 50 milliseconds would be too small for dynamic task assignment; Design A would have to be employed instead.

5.4 The Formulation of Policies

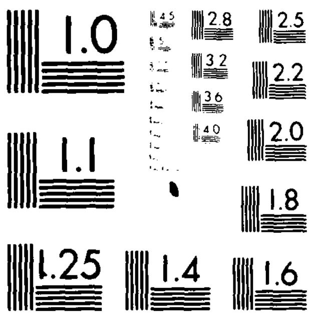
A policy is an expression of intent regarding the allocation of scarce resources among processes.

It is strongly related to the external goals of the system as a whole and the perceived relative value of the work performed by individual processes. In this sense policy formulation treats the computer system as a black box, and whether or not the system is physically distributed is not significant. For example, a

AD-A113 172 BOLT BERANEK AND NEWMAN INC CAMBRIDGE MA F/G 9/2
DISTRIBUTED OPERATING SYSTEM DESIGN STUDY. VOLUME I.(U)
JAN 82 H C FORSDICK, W I MACGREGOR F30602-79-C-0193
UNCLASSIFIED BBN-4674-VOL-1 RADC-TR-81-384-VOL-1 NL

3 OF 3
AD
A113 172

END
DATE
FILMED
5-82
DTIC



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

policy can establish priority for one class of users over another, and a consistent interpretation of priority can be developed for distributed or centralized system implementations.

Policy formulation for computer systems has much in common with policy formulation in society. The task of capturing the intuitive concepts behind a policy and expressing them precisely and completely is extremely difficult. Even more difficult is the assessment of the relative worth of user activities (users generally have strong personal or group biases) and the establishment of priorities.

Complex policies tend to evolve over time. Because policy formulation is usually based on case analysis complex policies have lengthy expressions. Policies are often expressed implicitly by the mechanisms which implement them. This approach presents a clear danger, in that interactions between cases may not be fully understood in advance, and procedures must exist for resolving conflicts. For instance, a policy which simultaneously seeks to grant high priority to interactive jobs and low priority to jobs with large memory requirements must also decide the priority of an interactive job with large memory requirements. All too often, the resolution of such conflicts is left to a system programmer and never publicly documented.

At the same time that a policy must be intuitive and complete, it must be feasible or capable of implementation. A feasible policy can easily become infeasible as it evolves through the addition of cases and constraints.

The feasibility requirement forces policy makers to understand the structure and performance properties of the system implementation, and it is at this point that the distinctions between centralized and distributed systems become important. The following paragraphs develop specific issues in policy formulation as applied to resource management in computer systems.

5.4.1 Objective Functions and the Concept of Utility

A substantial body of theory has been developed concerning linear and non-linear optimization. In this field, problem statements are usually of the form "maximize (or minimize) an objective function $F(x_1,...,x_n)$ subject to a constraint on the x_i given by the predicate $P(x_1,...,x_n)$." When the objective function F and the constraints P are both linear functions of the x_i, the problem is in the domain of <u>linear programming</u>. Practical solution techniques are available for very large linear programming problems. When the objective function or the constraints are non-linear the problem becomes more difficult, and fewer solution techniques are known. The problem statement above can be viewed as a policy, for which the value of the objective function provides a measure of utility or worth for a given system state.

The policy requires that the system be "operated" (the x_i chosen) so as to maximize its utility. This approach to resource management is applied to computer systems, most visibly in long-term planning. The utility can be defined directly as the reciprocal of cost, and an objective function can be composed from costs associated with values of the parameters. For example, the problem of determining the optimal placement and capacities of leased communication lines can be formulated as a non-linear programming problem. Optimization procedures are used for long-term planning and "static" allocation decisions, decisions which change only very slowly with time. The computer and human time required to solve the problem will be insignificant compared to the potential savings from even small improvements in a large system.

The optimization approach does [illegible] :tend easily to time-varying functions. It is difficult just to state simple intuitive policies such as "maximize throughput" when the constraints vary with time, and impractical to apply automatic solution techniques. Less direct policy statements that leave the numerical utility of system states unspecified are easier to construct and more intuitive; they prevent the use of numerical optimization techniques, however. The next section discusses the elements of such policies.

5.4.2 The Elements of Policies

Resource management policies are comprised of three types of elements: external administrative goals, internal administrative goals, and users' goals. We distinguish administrative from user goals because administrators are able to judge the relative value of user tasks and system-wide resource allocation decisions, while users are not.

5.4.2.1 External Administrative Goals

An administrator may know for reasons beyond the purview of the computer system that a task initiated by user A is more important than a task initiated by user B. If so, the administrator may assign A _priority_ for access to resources over user B. Priority may or may not be _preemptive_, so that if B's task is in progress when A's is initiated, B will be immediately deprived of the resources A needs to proceed. When a resource cannot be preempted, adherence to a priority scheduling discipline is limited to grants issued to queued processes.

Less severe policies for expressing biases are _pie-slicing_ (discussed in Section 5.3.7.1) and _class of service_ schemes. A class of service scheme categorizes a user task as belonging to exactly one service class, for example real-time, batch, or interactive. A class is granted resources based on some assumed common behavior of its members; tasks within a class are usually treated equally. Classes may be defined by implicit properties of tasks. For instance, processes which have accumulated less than 100 milliseconds of processor time since their last interaction may be placed in a high priority service class, and those with more than 100 milliseconds in a low priority class.

A resource management mechanism based on time-varying resource entitlements was proposed in [3]. This mechanism permits the relative utilities of processes to change as they consume resources, and thus represents a general setting for policies like the one in the last paragraph.

5.4.2.2 Internal Administrative Goals

Focusing on the cost-effectiveness of a system rather than external priorities generates policies to maximize resource utilization. Economic realities force some consideration of capacity planning at all installations.

A typical example of a policy based on cost-effectiveness is shortest-job-first (SJF) processor scheduling. Under suitable assumptions it can be proven that SJF minimizes the expected waiting time in a closed system (one in which the number of processes is fixed), and maximizes the throughput in a saturated open system (one in which the number of processes is variable). It minimizes waiting time, however, at the expense of increasing the variance of the response time; large jobs are delayed longer.

5.4.2.3 User's Goals

Individual users are naturally concerned about receiving enough resources to carry out their assigned tasks. Users' claims for resource allocation can be roughly characterized as:

Subsistence	the minimum level of resource grants necessary for a user to accomplish a task;
Fairness	in the absence of external constraints, one user is entitled to as much of a resource as another;
Quality of Life	users need some resources above the subsistence level in order to work quickly and effectively.

Scheduling algorithms often attempt to achieve fairness for some class of customers. The round robin approximation to the processor sharing discipline is one example of a fair scheduling policy. Quality of life arguments are visible in the temporary increase of priority given to processes immediately following user interactions.

5.4.3 The Interplay of Goals

The resource management policies of a system are usually formulated from several goals as described above. It should be recognized that the goals are almost always in conflict. There is a clear tradeoff between imposed priorities and fairness to users; sufficiently harsh priority structures impact the user quality of life, and eventually subsistence.

Every timesharing user is familiar with the cost-effectiveness versus quality of life tradeoff, in the form of poor response time when the system is heavily loaded. The tradeoff between priority and resource utilization is perhaps not so clear, but can be very important. A rigid priority structure prevents scheduling algorithms from making some decisions which would increase resource utilization. The effect can be dramatic in single host systems, decreasing processor utilization on the order of 50%. When resources are scarce there is a tendency towards priority-based policies, with stronger administrative control. Very scarce resources may be managed statically (through human planning) just to avoid the overhead in automatic management. When resources are abundant, fairness and quality of life arguments gain influence.

The issues discussed in this section are primarily organizational. It is difficult to imagine a computer system with a pervasive priority mechanism run by an organization lacking one. Without centralized administrative control over system resources, global policies cannot be implemented.

5.5 Resource Management Issues in Distributed Systems

Resource management in a distributed system is distinguished from resource management in a conventional centralized computer system by three principal factors:

1. Distributed systems are often vastly more complex than centralized systems, and can have hundreds or thousands of allocatable components.
2. Since distributed systems contain multiple processor and storage components, there is a possibility of improved reliability through redundant data storage and the reassignment of tasks to healthy processors.
3. The message transmission time between processes on different processors can be large, relative to the message transmission time between processes on the same processor.

The first factor suggests that the (non-automated) administration of the staff and physical facilities of a large distributed system may be difficult. This directly impacts the scope of automatic resource management decisions within the system, since any resource manager must have authority to control the resources it manages. Pre-existing organizational frameworks may restrict the ability to centralize authority, and thus limit the effectiveness of automated resource management strategies. For example, a distributed system host "owned" by a particular agency or department may not be available for permanent file storage to other agencies or departments, even though the system could be incorporated into a global allocation scheme. When authority over physical and logical resources (i.e., databases) is fragmented, the utility of global resource management should be questioned.

Redundant data storage and reliability mechanisms complicate global resource management because they interact with it. The need to maintain two copies of data on storage devices with independent failure modes, for instance, clearly complicates the allocation decisions for the device resource manager. The use of task reassignment to achieve survivable operation can be applied to client tasks and tasks within the DOS itself; in particular, it is highly desirable for the DOS resource managers to be survivable in order to provide continued service with the remaining system resources to DOS clients.

Message delays have important implications for the structure of global resource managers. The essential fact is that local (independent) decisions can be made more rapidly than global (consensus) decisions, in the absence of prior knowledge about demand patterns. When the communication paths are very slow, most of the resource decisions must be made independently, and only long-term scheduling can be done globally. With faster transmission mechanisms the burden of scheduling can be shifted toward global strategies.

The following sections expand upon these ideas, first from the DOS client's viewpoint in Section 5.5.1 and then from the global resource managers viewpoint in Section 5.5.2.

5.5.1 The DOS Client Interface

5.5.1.1 Visibility of Distribution

To the extent that the distribution of resources is visible to DOS clients and under their direct control, the DOS is prevented from global resource management. If it is possible for a client to specify that a file be allocated on a particular host, or that primary memory on a particular host be dedicated to a process, the DOS is unable to make allocation decisions for these resources. Only when a client requests one or some units from a pool of identical resource units is global resource management possible.

During the DOS design, fundamental decisions must be made concerning the resources which will be globally managed by the

DOS. In the past, the highest priority for distributed management has been accorded to the shared objects central to the IPC facility (i.e., "sockets", "ports" and "connections"). Some systems include a distributed file system and the required storage devices within the scope of the DOS [8], and a few manage processor resources [13]. The processor resource is difficult to manage globally because the event rate (process activation-deactivation rate) is higher for a processor resource than, say, for a file resource. Systems which attempt to dynamically assign tasks to processors are likely to be built around high bandwidth communication media; this is an example of the grain phenomenon discussed in detail in Section 5.3.8.

Resources may be placed under client rather than DOS control for a variety of reasons. The topology, bandwidth, and propagation delay of the communication network may force DOS client programs to exercise precise control over task and file assignment to meet performance goals. Generally, the visibility of distribution to a client will increase as its demands approach the ultimate performance of the underlying hardware. In contrast global resource management becomes more attractive as the number of clients increases, each requiring only a small fraction of the available resources.

Clients may be aware of the distribution of resources because the administrative authority for their management is distributed. This is true, for example, of computer accounts on different hosts within the ARPANET; an ARPANET user must (potentially) negotiate with a separate organization (business, agency, university) for access to each ARPANET server.

Depending upon the facilities provided by the DOS, the need for synchronization of updates and accesses to a database by different clients may force knowledge of the location of the database; in particular, it may force a unique copy of the database to be maintained.

5.5.1.2 Resource Subsets

A resource manager has a pool of resource units under its control which it uses to satisfy the clients' requests[14]. In principle all units of a resource in a distributed system could be controlled by one resource manager, and be equally accessible to requesting clients. Situations often dictate separate managers for subsets of the pool of resources of a given type, however. Similar resources may belong to different resource

[14]The allocated resources may be preemptable, in which case the manager's control extends through the period of allocation.

managers because:

1. Under close examination, the units of the resource are not precisely identical.

2. The topology of the communication network forces "regional" resource management to minimize communication delays.

3. Administrative controls (e.g., security) influence the permissible sites for processing or data storage.

4. Unreliable equipment is excluded from a resource pool even though it is otherwise acceptable.

5. It is expensive to extend the implementation of a resource manager to a new host supporting units of the managed resource.

Resource subsets are usually directly visible to client processes, and clients must invoke different resource managers to obtain resources in separate subsets.

An example of the first factor is the incompatibility of processor types within the ARPANET. A program written for a DECSystem-20, say, is almost certainly incompatible with an IBM host, and could be incompatible with a DECSystem-10 host or even a DECSystem-20 host running an earlier release of the monitor. Depending on the requirements for compatibility, the resource pool available to that program may have to be severely limited. (Since dynamic assignment of tasks to processors is not the normal mode of operation on the ARPANET, the problem does not occur in practice.)

The division of resource units of the same basic type into separate resource pools is itself a resource management decision on a relatively long time scale. Clients are often restricted to obtain the resource units they need from one manager, and thus the allocation of units to pools will affect the rate of progress of client processes. Too many managers for one resource may make it impossible to enforce a coherent global resource utilization policy.

Resource subsets are closely related to the issue of binding in a distributed system. A _binding_ links a _name_ with a _value_, and is one of the most fundamental concepts in computer systems. In centralized systems, binding issues are simplified because there are fewer bindings to construct, and many of them can be formed statically. Distributed systems require the maintenance of many more bindings for their operation, between names and users, processes, hosts, files, ports, etc. The creation and deletion of bindings is more difficult in distributed systems because the information relevant to the states of bound objects may be scattered among several hosts. Binding has been discussed earlier from the perspective of reliability.

The construction of resource subsets is also closely tied to the problems of "file assignment" and "communications capacity".

Considerable work has been done on these problems in isolation (see [53, 43]). The most commonly used approach is through network theory, and linear or non-linear programming. Static allocation decisions (in reality, resource allocation decisions with a very large grain) determine resource subsets available for automatic allocation within the distributed system. Static allocation decisions are very important when the communications bandwidth is small, and decline in importance as the bandwidth increases and dynamic reconfiguration becomes less costly.

5.5.1.3 Performance and Reliability

To some extent reliability and resource control are at odds. Reliability mechanisms depend upon the maintenance of synchronized, redundant copies of information, whether in the form of process checkpoints or redundant databases. As a natural consequence they place an added burden on the resource allocation mechanisms, both because they increase the demand for system resources and because they introduce a new dimension along which resources must be distinguished by managers, namely, the independence of failure modes. Client processes attempting to achieve reliable operation may request redundant storage areas, for example, that are identical in all respects except for their failure independence; this aspect must be recognized and catered to by the responding resource manager(s).

Reliability mechanisms may also complicate resource control for the client by introducing a new level of abstraction between the client's view of DOS objects and the physical resources of the distributed system. Ultimately, the usage of physical resources determines the performance properties of the client programs; in order to optimize the performance of a client program, it must be possible to comprehend the relationship of DOS interface operations to the consumption of physical resources. The introduction of new concepts for reliability such as stable storage [32] makes this more difficult. Some aspects of reliability mechanisms that should be carefully analyzed for their impact on resource management are synchronization, recovery operations, and the allocation of resource units with independent failure modes.

5.5.1.4 DOS Operations for Resource Control

Resource management decisions cannot always be made automatically, and it is important that DOS clients be able to influence or control some aspects of resource management. The central design decisions concern a precise definition of the set of resources to be controllable by a client, and the manner control will be exercised. For example, clients might be permitted to control to some extent the memory management strategy applied to them (clients might be able to choose between demand paging, swapping, and swapped working set policies) but be unable to control the degree of buffering accorded interprocess messages. The client gains leverage on performance problems by being able to influence resource allocation, but the DOS becomes more complex and administrative control over resource policies may be diluted.

If a DOS client is part of a distributed task there are other issues to be resolved. In general a mechanism must be provided for transferring resource control authority, initially from the DOS to a client process, and subsequently from one client process to another. Decisions must be made regarding the grain of resource control to be exerted by a client process. It may be desirable to limit the topological scope of authority. For example, the influence of a client process over resources may be restricted to adjacent resources (i.e., those residing on the same host as the process) or those resources in the same cluster, where the high interprocess communication bandwidth permits a fine grain of control. Thus we envision clients in a local cluster participating in the scheduling of local host processors, but not participating in the scheduling of processors belonging to remote clusters because of the sizable communication delay.

Because a client becomes a resource manager when it asserts control over system resources, the issues in Section 5.5.2 are pertinent. In particular, the failure of a client process (whether due to hardware failure or an incorrect program) may cause resources to be temporarily or even permanently lost to the DOS, if the client had authority for the allocation of the resources when it failed.

The design principle in Section 5.3.8 limits the grain of resource allocation decisions when the demand patterns of DOS clients are not known in advance. This limitation can be very severe when communication delays are large; it can be avoided, however, if the demand patterns are known in advance. In this case <u>reservations</u> can be employed to allow fine grain coordination, in a manner described below.

The success of a reservation strategy depends upon the existence of accurate, synchronized clocks at remote hosts. We assume the hosts normally communicate over channels with relatively long propagation delays and high variance. Suppose a distributed transaction must obtain resources at three hosts A, B, and C, simultaneously, and that A is the originating host. At a time T1, A negotiates reservations for the needed resources with B and C, the resources to be committed at a time T2, where T2-T1 is large compared to the negotiation time. At time T2 the resources are simultaneously allocated at all three hosts, and the transaction is initiated by A. When the transaction proceeds from A to B and C, it will find the resources it needs already allocated and waiting.

Reservations are used for some purposes in conventional systems, for example, users ar3 often allocated disk capacity in anticipation of future need. The use of reservations in global resource management has not been thoroughly studied.

5.5.2 The Implementation of Managers

The implementation of a manager for a distributed resource may be centralized or distributed. Both techniques have been employed in the past. The National Software Works concentrates many allocation decisions in the Works Manager process bound to a single, distinguished host in the network. The routing algorithms used by ARPANET packet transmission nodes represent the fully distributed extreme, with nodes cooperating in a loose way to allocate line capacity and buffer space where it is most needed.

5.5.2.1 Centralized Manager Implementation

The advantages of a centralized resource manager implementation are:

1. The manager can be constructed with proven, sequential

program techniques.

2. Message traffic may be reduced, since the manager need not use interhost messages for internal purposes.

3. Administrative policies may be easier to enforce, since policy is concentrated at one site.

4. The manager may be easier to construct and maintain because code is only developed for one computer system.

The major disadvantage of a centralized resource manager is the sensitivity to a single failure point. Reliable operation can be achieved by appointing "heirs" to the resource management function, existing on alternate hosts. In the event of a failure, one heir is chosen by some means to become the active resource manager, and the others remain in the passive state. This approach was utilized in the NSW Reliability Plan [38].

5.5.2.2 Distributed Manager Implementation

Distributed resource managers are inherently more complex than their centralized counterparts. Concurrent programs are usually more difficult to construct, test, prove, and modify than sequential programs. A distributed resource manager may possess some compensating advantages, however.

1. If the manager is properly designed, survivable operation may be a natural consequence of a distributed implementation.

2. The manager may be constructed so that the throughput of allocation decisions scales with the number of participating hosts.

3. Because of distribution, the manager may more easily participate in scheduling activities local to the hosts which support its component parts.

Point (3) suggests that a closer integration between local and global scheduling may be possible if the implementation of the resource manager is distributed.

The disadvantages of a distributed implementation, in addition to the problems of writing concurrent programs, concern the extra messages needed to transmit state information between resource manager components, the difficulty of capturing a "snapshot" of system state for debugging or recovery, and possibly the need to construct manager components under different host architectures.

5.5.2.3 The Maintenance of Resource State Information

A resource manager may choose to maintain state information which is accurate at all times (up to communication delays), or it may update its internal state only at allocation/deallocation events, or it may adopt a compromise. A tradeoff will normally exist between maintaining up-to-date information (permitting rapid allocation decisions, but requiring additional message exchanges to distribute state information) and reducing the overhead incurred by the manager (allocation decisions may be delayed while accurate state information is obtained, but fewer messages need be transmitted).

Several systems have employed the update-on-demand technique. In the RSEXEC system, for example, TIPS broadcast a "Can you accept login?" message to a set of candidate hosts when a user attempts to login. The first responding host is selected, on the hypothesis that it is likely to be less heavily loaded than the hosts responding later. A more cautious approach might be to include the system load average in the host to TIP reply, so that the TIP could choose the most lightly loaded host with high probability.

The routing tables in the ARPANET packet switches mentioned previously are an example of the continuous update approach. In this system status information is continuously exchanged among switches. The ARPANET switches keep the overhead to a manageable level in two principal ways: First, it is sufficient for a switch to use slightly out of date data and as a result make allocation decisions that are short of optimal. Second, some of the status information needed by the switches is piggybacked onto routine traffic, and thus represents only a small additional network overhead.

5.5.2.4 The Transmission of Authority

The authority of a client to obtain a given resource, in the absolute sense, and its priority for obtaining the resource, in the relative sense, must be stored as protected state information within the computer system. This information may be centralized or distributed, stored within resource managers or in some other repository. Since managers must first determine the authority of a client to obtain a resource before it can be allocated, this information must be readily available if allocation decisions are not to be delayed. This suggests that the permission information follow the manager implementation -- a centralized manager can most easily utilize a centralized permission database, and a distributed manager can benefit from distributed permission information (provided the distribution is carefully chosen).

5.6 Strategies for Global Resource Management

This section presents two examples of global resource management strategies, based on statically assigned priorities and time multiplexed resource consumption. Issues concerning both policies and mechanisms are discussed, but the treatment is by no means exhaustive.

5.6.1 Static Priorities

5.6.1.1 Priority-Based Policies

When we say that process A has priority over Process B, we mean that an authority external to the computer system has judged the work performed by A to be more important than the work performed by B. The authority dictates that system resource managers give preference to the resource requests of A over those of B, when A and B compete for the same resources. Unless some external authority has this control over the behavior of resource managers, priority scheduling is impossible.

Preference may be given to a high priority process with or without <u>preemption</u>. With preemption, a resource manager will preempt or reclaim immediately resources needed by the high priority process and currently allocated to a lower priority process. Without preemption, processes may hold resources they have been allocated until they voluntarily release them. In either case, processes waiting for resources in a manager's queue are serviced in priority order as the resources become available.

Preempted processes must either be suspended until the resource becomes available again, or they must be notified of the preemption through an exception mechanism. Interrupt handling is an example of the former case; an interrupt causes processor state to be saved in some area of memory, the interrupt handler to be executed, and the interrupted process to be resumed. The practicality of the preempt-resume discipline in any given situation depends on the existence of the state save and resume operations, and on the cost of the operations. Preemption is quite practical, generally, for processor scheduling; useful over longer time periods for memory scheduling, because the preempt and resume costs are higher; and not useful for processes reading and writing magnetic tapes, because it is not feasible to save and restore the intermediate state information existing only on tape.

Because priorities are set by external (usually human) authority they tend to be static, at least relative to the process activation and deactivation rates inside a computer system. Thus it is not practical to associate external priorities directly with processes, and instead priorities are assigned to users, projects, roles, departments, etc. These priorities are stored as a database (centralized or distributed) within the computer system. The system also incorporates rules which determine the priority of any process as a function of its relationship to the statically assigned priorities. For example, if the Superintendent is entitled to priority 3, processes initiated by the Superintendent will execute at priority 3.

Processes should be able to transmit their priorities along with service requests to other processes. The service module is then permitted to execute at the priority of the requesting process, while it is operating on its behalf. In fact, the capability model developed for Hydra [74] can be applied directly to the transmission and revocation of priorities.

5.6.1.2 Mechanisms for Static Priorities

A simple priority-based strategy might operate as follows:

1. Processes have a priority and an owner. When a user creates a new process through a direct interaction with the DOS, the user becomes the owner of the process, and the process priority is derived from the user's priority.

2. Whenever a process creates a new process, the owner and priority of the new process are taken as the owner and priority of the old process.

3. When a process A owned by one user transmits a message to process B owned by another user, A may authorize B to use its priority for a limited period of time. Process A specifies the expiration time in the message.

4. When process B receives a message from process A, its priority is set to the maximum of its prevailing priority and the priority authorized by A.

5. If process B reaches the expiration time specified in the last message which caused its priority to change, B's priority is reset to the priority of its owner.

Many improvements and customizations to this simple strategy for disseminating priorities are possible, but it should serve as an example of what might be done.

Once a definition for the priority of DOS processes is available, it remains to specify how the priorities should be carried out. Within a COS the answer seems to be clear-cut: the DOS assigned priorities should be treated by COS resource managers (for the processor, primary memory, disk channels, etc.) with complete trust. The same mechanisms used to implement priority scheduling within the COS suffice for DOS scheduling.

One shared resource is not managed by the COS's, namely, the communication network. Priority may be reasonably extended to priority for access to the communication network, and we give one example of how that might be done here.

The Ethernet [37] was the prototype for a class of local networks known as contention or broadcast networks. The strategy described below is adapted to utilize a contention network as the communications medium; in particular, we make use of the fact that any message transmitted can be overheard by all stations attached to the network.

1. Every message transmitted is stamped with the priority of the sending DOS client process.
2. All network stations receive the priority field of every message transmitted (they receive the remainder of the message only if it is addressed to them).
3. The network priority is defined to be the priority of the last message transmitted.
4. If the network priority is P and a DOS process with priority Q wishes to transmit, there are two cases:
    - o $P<=Q$: the process transmits, and the network

priority becomes Q;

- o P>Q: the process is forced to wait, until the network priority is reduced.

5. A DOS watchdog observes the network traffic, and if no activity at network priority P is observed after some time interval, the watchdog transmits an empty message with priority P-1, reducing the network priority.

An interesting refinement is possible with regard to the backoff algorithm used to delay retransmission attempts when a collision is detected. If two processes collide, their backoff intervals can be weighted by priority, higher priority having the smaller weight. It is thus probable that the higher priority process will attempt to retransmit first, and lock out the lower priority process by increasing the network priority.

The treatment of processes of equal priority has not been discussed, but some attempt at "fairness" is probably required. Whether "fairness" is necessary, and whether permanent blocking is inherent in the "fairness" concept, is an administrative issue rather than a technical one.

5.6.1.3 Priority and Utilization

Whenever absolute priorities are contemplated, their potential effect on resource utilization should be understood. If processes can hold several resources simultaneously, absolute priority can result in arbitrarily low resource utilization for almost all resources, in the worst case.

For example, suppose a uniprocessor with disk storage is executing one task, which consumes 50% of the processor and 50% of the disk channel capacity. A higher priority task enters the system, and this second task computes but does no I/O (or only a negligible amount); the high priority task consumes 100% of the processor but forces disk utilization to zero.

A similar situation could occur wherein priority for the disk channel forces processor utilization near zero.

5.6.2 Time Multiplexed Tasks

5.6.2.1 The Policy as a Form of Reservation

The technique of _time multiplexing_ divides the access periods of several processes to a resource into _slots_, discrete intervals sequential in time. Usually the pattern of slots is repeated cyclically many times, and changes only slowly relative to the slot duration. An important example of time multiplexing in computer systems is processor multiplexing, which makes one physical processor appear to be many virtual processors. The technique is also commonly used in communication systems, where each slot represents a _virtual circuit_ or _connection_. This approach is valuable in part because of the convenience of _time_ domain switching, i.e., switching circuits by permiting the order of slots from an incoming to an outgoing data stream.

Many variations on the basic concept are possible. Slot durations may be fixed or variable, and control over the slot assignments may be centralized or distributed. In the centralized case, the slot duration can be apportioned by the central contrcl to reflect the entitlements of processes to the resource, i.e., more valuable processes receiving larger slots and consequently better service.

The Flexible Intraconnect (FI) being developed for the Tactical Air Force will permit the communication network to be time multiplexed in very flexible ways. In particular, there are two modes of operation known as Virtual Bus Modes A and B that permit variable and fixed duration time slots, respectively. The FI will support up to 63 Virtual Buses simultaneously, and each can operate in either Mode A or B, the mode being fixed at system initialization time.

We envision extending the time multiplexing provided by the Virtual Bus to include time multiplexing COS resources. Under the FI VB Mode B, for example, fixed time slots are allocated to particular interface units; the DOS could coordinate these tim slots with COS memory management by swapping DOS tasks into memory slightly in advance of the slot, and locking them until their time slots expire. This mode of operation would permit the overhead of swapping operations to be overlapped with other COS activities, and would seek to achieve the greatest degree of resource coordination within a distributed task.

In essence, this approach considers a time slot to be a reservation for future service. It relies upon the assumption that distributed tasks will complete more rapidly if they are able to obtain all needed resources (including communication access) at the same time.

5.6.2.2 Mechanisms for Time Multiplexing

To achieve globally synchronous operation as described above, the distributed DOS components must be able to accurately compute the positions of time slots. This can be done by maintaining stable clocks at each COS which are synchronized only rarely, or less stable clocks that must be synchronized more often, or the synchronization can be done entirely by messages transmitted at the beginning of each slot time. Any one of these approaches may be practical under particular circumstances.

The DOS components at each COS must be prepared to allocate COS resources in time for upcoming slots. This means that slot times cannot be too small, or the hosts will be unable to complete the allocations. It probably means that only preemptable resources can be allocated in this way, and that the time required for preempting (e.g., the time required to swap out a process) must be on the order of the slot time, or less.

Although we are unaware of serious attempts at resource allocation strategies of this type, we believe they are worthy of serious consideration at this time. The high bandwidth available through local network technology reduces the grain of scheduling decisions that can be accomplished in a distributed system. The Flexible Intraconnect provides features to support mechanisms of this type that have been previously unavailable.

5.6.2.3 Clock Synchronization

The ability to build stable local clocks and accurately synchronize them is important to reservation based resource management schemes. The technology for building extremely stable local clocks is well known and not particularly expensive; the problem of synchronizing the clocks is more difficult.

If clocks are to be synchronized by means of messages transmitted through the network, the variance of message delivery times is a fundamental limit to the precision achievable. Therefore it is useful to understand the propagation delay and variance for a local network, and if possible, to take measures to reduce the variance. For example, a special operating mode might be used to synchronize system clocks, in which no messages except synchronization messages from a central clock are permitted on the network.

Clock synchronization is a fruitful area for further study in distributed systems [31].

6. COS FEATURES FOR EXTENSIBILITY

Experience suggests that some operating systems are much easier to incorporate into a DOS than others. It is natural to inquire as to why this is so, and what features or capabilities should be present in a Constituent Operating System (COS) for it to be easily assimilated and an efficient base for the DOS functions.

The hosts connected to a distributed system can be classified into three types:

1. Medium or large scale shared hosts.
2. Special purpose processors.
3. Personal or dedicated general-purpose computers.

The first type is a multi-user system which may supply services to a number of clients directly attached to it or accessing it through the network. The second type includes processors such as the image correlation unit in the FCU example of Chapter 3, which are dedicated to very specific uses and have limited programmability. The third type includes single-user-at-a-time processors with general programmability, but no shared access; the workstations of the Air Traffic Control example are assumed to be of this type. This chapter is directly related only to the first type, shared hosts, and addresses the characteristics of the COS for shared hosts; it may have some implications for the third type.

We assume that the DOS functions are to be constructed on a logical level above those of the COS, and it is desirable to make minimal changes to the COS implementation in developing the DOS implementation. The DOS processes are clients of the COS's in the distributed system, and obtain local services by requests made to the COS's. A COS may have additional local clients that do not participate in the DOS at all, and in this event we expect some provisions for preventing or minimizing accidental interactions between the client communities. What primitive services should the COS offer to its clients? The sections below explore the question.

An extension to an operating system is the addition of any software or hardware that enriches some client's view of the functionality of the system. A new software product in the form of a compiler or database management system is an extension that would probably be visible to many users; functions defined in a private library of functions and procedures may be extensions visible only to one client or a small group of clients.

The fundamental means by which a system is extended through software is by the addition of data structures to the permanent storage media of the system. These data structures may include programs, static data, and data that changes over time. From this viewpoint an _extensibility mechanism_ is any OS feature for the creation of long-lived structures. The three major categories of extensibility mechanisms in operating systems are:

1. Program structuring
2. Directory services
3. File access methods

Program structuring mechanisms enable a programmer to divide a program into smaller pieces or modules which are in some sense "separate". Because program structuring occurs at many levels from firmware to high level languages it is difficult to state generally what constitutes a module, or how modules combine; in a specific context the concept is usually clear. A few classical program structuring mechanisms include the subprogram or procedure facility, separate compilation, separate processes, and the utilities for linkage and cross referencing that accompany them. A particularly important type of program structuring facility for distributed computation is the solution of a complex problem by multiple cooperating processes.

Directory services are the means for associating names with operating system objects. Names are usually character strings, and the objects referred to are typically files, users, processes, devices, hosts, or directories. Because a directory can be used to represent many-to-one relationships (many names for one object) and because directories themselves are objects and can be named, very complex structures (arbitrary graphs) can be represented by a general directory mechanism. Although most practical systems do not provide full generality and place restrictions on the connectivity of directories, this structuring mechanism remains extremely important and useful.

A file access method is a means for efficient storage of data on secondary memory devices. Access methods can be extremely simple, permitting only sequential access to a data file through OPEN, READ, WRITE, and CLOSE primitives, or extremely complex permitting associative access to data items and managing a hierarchy of storage devices (e.g., staging data from tape to disc). File access methods support a client's logical view of data and aid in the construction of databases.

Different operating systems emphasize different areas of extensibility. IBM's OS/VS offers a wealth of file access methods but is relatively deficient in primitives for managing processes; in the UNIX operating system the situation is just the reverse. The result is that some kinds of extensions are easier to build and possibly more efficient on one system than another. We expect that it is easier to construct database systems on OS/VS than on UNIX and easier to develop communication software on UNIX than OS/VS.

The areas in which an operating system is readily extensible will usually correspond to mechanisms recognized by the operating system designers as central to the system concept, and subsequently carefully developed into general, efficient, convenient, well-documented, and stable OS features.

6.1 COS Design Versus Retrofit

We believe that the extension of a COS to a DOS host places heavy demands on COS extensibility. The demands might be met by the design of a completely new COS or by the selection of an existing OS and its modification to achieve the necessary flexibility. These two points of view dictate different approaches to the problem.

When a COS is designed from the beginning, the opportunity exists to choose central system concepts to be compatible with the expected use of the system. For instance, a COS designer might choose to provide a very powerful process concept with long-lived processes existing on secondary storage, a strong interprocess communication facility, and reliability mechanisms for checkpointing processes. Efficient implementations are possible for the concepts if this is a major design goal. Some less important system features may suffer as a consequence of the emphasis on COS extensibility, but the choices surrounding the tradeoffs are under the control of the system designers.

The person selecting an operating system to become a COS is faced with a different problem. For most commercial machines there are a small number of major operating systems available in any case, and the selection must be made from among them. None will be entirely satisfactory, and the choice is likely to be made as much on the basis of familiarity and compatibility with existing software and hardware as on isolated technical merit. Once the OS is selected, some modifications will have to be made to adapt it to the distributed environment. Because operating systems are complicated and modifications are costly and time-consuming there is pressure to keep the changes to a minimum. Major system concepts of the selected operating system will not be changed; components such as the processor scheduler and virtual memory manager may be tuned, but will probably not be redesigned.

Some of the discussion below applies only to the case where one contemplates a new beginning, with the possibility of a strong departure from the past. Other features can be applied to existing systems with some potential benefit.

6.2 COS Extensibility Needs

In this section four areas of operating system extensibility we feel are particularly important in a COS are discussed. They are:

- o Process structuring and messages
- o Process and file directory services
- o Resource control and accounting
- o Client authentication and access control

6.2.1 Process Structuring and Messages

Distributed systems are necessarily composed of multiple processes, at least one per processor node. We have observed that once processes are acknowledged as a unit of modularity they are exploited for this purpose even on a single host and within a single application. In testing phases there is a natural tendency to use a process on a local host to mirror or represent a process on a remote host with which it communicates.

Recently economic and technical forces have caused increased interest in process structured application programs. Three prominent examples of this trend are the National Software Works, a distributed operating system [8], INGRES [61], a relational database system, and Hearsay-II [12], a speech understanding system. Each is composed of several cooperating processes running as OS clients. An interactive request to one of these systems may activate a substantial number of processes and cause a flurry cf interprocess messages. In the case of NSW the messages may travel over the interhost communication network.

The trend toward process structured application programs will continue and intensify. The declining cost of processors and the availability of inexpensive local networks encourage the use of real parallelism at the level of processor modules; the introduction of parallel programming constructs into high level programming languages such as Concurrent Pascal [6] and Ada [27] encourage the use of virtual parallelism as a structuring methodology. In the case of COS design, where communication between asynchronous processes on separate COS's is expected to be commonplace, facilities for process structuring are vital.

There are several areas of concern for COS process structuring. A COS client process should have the ability to create new processes dynamically at low cost. The processes will cooperate by means of Interprocess Communication (IPC) that allows them to exchange messages. For high bandwidth transfers exceeding the capacity of the IPC, it is desirable for processes on the same COS to be able to share memory regions.

Process structuring is an important modularity mechanism in a COS, and thus a COS should contain a rich set of primitives for manipulating processes. At a minimum system calls should allow an executing process to:

1. Create or destroy other processes.
2. Suspend execution for a fixed time interval, from fractions of a second to days, weeks, or months.
3. Interrogate the status of other processes in the system (does a process still exist? is it dormant?).
4. Suspend the execution of other processes, and later explicitly resume them.

The number of processes which can be created should not be limited by any small, fixed-size system table. Processes which are dormant should cause very little additional overhead, in particular, processes dormant for a long period of time should not require any primary memory space during the period of inactivity. It is desirable to treat a dormant process in much the same way as a data file, checkpointing it periodically and possibly permitting the storage of processes on demountable media (disk packs and magnetic tape).

Message primitives serve a dual role, transmitting data objects and defining synchronization points between processes. A summary of recent proposals for message primitives can be found in [25].

The issues surrounding interprocess communication center on these factors:

- o The way a logical binding is achieved (partner named) for the evaluation of a send and receive primitive pair.
- o The amount of message buffering to be provided automatically by the system.
- o The syntactic and semantic compatibility of various message facilities with high level programming languages.

It is generally recognized that processes must be able to wait for messages from one of several different sources simultaneously. There is also a need for a time-out mechanism to terminate a send or receive operation which does not complete after a specified time interval.

The structure of message based systems is still under active research, and because it is a not a primary responsibility of this work will not be pursued here. More information on process and message structuring can be found in the references [25, 27, 15].

6.2.2 Directory Services

Directory services are the means by which name-to-object bindings are established. A centralized system normally maintains one copy of each object (e.g., process or file) and thus the name translation process is fairly straightforward. In a distributed system, however, the high cost of communication between sites motivates the replication of objects at multiple sites. The binding mechanism can be greatly complicated by the existence of redundant copies of files and processes, the need to locate a copy at binding time, and the need to restore bindings in the event of failures.

If the directory services of a COS are sufficiently flexible the COS clients may be able to modify and use them for their own needs. An important requirement is that long object names be permitted, to allow clients to superimpose their own structure on the COS directory services. It is also useful for directory entries to have a provision for client defined attributes (such as a type field). It is difficult for the COS designer to know in advance how much flexibility to supply; traditionally clients are not able to encode information in directory entries except in the entry name itself. It has been proposed to structure all of the operating system information, including directories, in a general database system [68]. Through the database mechanism client processes could construct alternate views of the system tables in the COS, and add or delete information from their private views as necessary. This approach has not been given a full-scale test to our knowledge, but merits further study.

6.2.3 Resource Control

One of the major responsibilities of an operating system is the management or scheduling of shared resources among its clients; this is equally true of distributed and centralized operating systems. A DOS, however, must construct its resources from those of the underlying constituent hosts, and must cooperate with the COS's closely. Contemporary operating systems make extension of resource management into the sphere of client processes difficult or impossible.

As an example, imagine a DOS with two distributed client processes, DC1 and DC2. At the DOS level an administrative decision is made to allocate 90% of the available primary memory to process DC1 and 10% to DC2.

In order to implement the policy, the DOS must obtain some knowledge of the availability of the primary memory resource within COS components, and be able to influence COS memory management decisions in the appropriate manner. Typical operating systems available today either prohibit a client process from influencing another client's memory allocation, or permit it only if the client is granted blanket authority to control memory allocation for any client process. Neither situation is acceptable for the needs of a COS.

The research operating system Hydra [34, 73] developed at CMU for a multiple PDP-11 configuration known as C.mmp embodied many of the resource control ideas we feel are important in a COS. Hydra has provisions for special client processes called _Policy_ _Modules_ (PM's) that are able to schedule the processor and memory resources of other clients in a controlled way.

The important features of Hydra as a model for COS resource control are:

- o The ability of client subsystems to appoint a Policy Module to manage resources for the group.
- o The separation of short and long term scheduling in order to reduce overhead.
- o The presence of resource guarantees that lead to predictable performance properties of subsystems.
- o The existence of capabilities for PM status and policy objects, preventing interference of subsystems over resources.

Similar features could be included in a COS for a hardware environment more conventional than C.mmp.

6.2.4 Authentication and Access Control

User authentication and access control are services likely to be offered by a DOS. They are also services that will probably be present in a COS for the use of COS clients. It is desirable that the DOS functions for authentication and access control be obtained by extending the COS functions, rather than by an entirely new set of mechanisms implemented at the DOS level. Extension is preferable to separate implementation for several reasons. Software development costs may be reduced simply because the DOS implementation is smaller. Security is improved because fewer lines of system code are involved with authentication and access control, and thus can be more readily verified. System performance may be improved by the elimination of interpretive evaluation of access controls at more than one system level (e.g., evaluation of a DOS file protection code followed by evaluation of the COS protection code for the COS file representing the DOS file).

In order to realize these advantages the COS must be prepared to cooperate with its clients in the proper way. A capability based system such as Hydra permits almost unlimited flexibility in the construction of alternate protection regimes by client processes, and is a nearly ideal environment for extension. No widely available operating system appears to implement the full generality of the Hydra capability scheme.

6.3 Enhancement vs. Elevation

We distinguish two types of extensions to computing systems, _enhancements_ and _elevations_. An enhancement is an extension which is visible to all of the clients of the extension. Usually when we consider extending a system the extension is, in this terminology, an enhancement. For example, the addition of a new subroutine to a program library is an enhancement of the library; in order to use the new subroutine, a calling program must be aware it exists (know its name, or a means for determining the name) and must understand its function (know its input and output parameters and side effects). Whenever a "new feature" is added to a computer system, we say the system is enhanced.

An elevation is an extension whose visibility is relative to the client using the elevation. The concept will be more fully explored using abstract types below. As an example, suppose a two level memory hierarchy in a typical timesharing system (primary memory and disk storage) is used by a client process P. P opens, reads and writes the files stored in the memory hierarchy. A standard set of operations is provided by the OS to perform these functions. A revision to the operating system is then generated which extends the memory hierarchy to three levels (primary memory, disk storage, and tape storage). The extension has been designed so that process P can continue to work as before, with no changes to its code, because the primitive operations used by P support exactly the same interface as before. There may be a new interface used by program Q but not used by P (because it didn't exist when P was written) which allows explicit control over the movement of files between tape and disc.

This extension we call an elevation, because in effect the functionality of program P has been "elevated" or "lifted" without its knowledge to utilize the new memory hierarchy. The elevation is invisible to program P, but visible to program Q[15] .

Although most functional extensions to computer systems are enhancements, elevations can play an important part in distributed systems. An elevation offers the opportunity for extending the function of a system component without altering it;

[15]an elevation which is invisible to all observers is not a true extension, since it is indi^tinguishable from the unextended system.

thus elevations are economically motivated. The implementation of the TELNET communications protocol in the ARPANET is an example of an elevation in a distributed system. Remote users of a computer system are aware of the TELNET link between hosts since they explicitly construct it; the programs used at the remote site are generally unaware of the presence of the link. The protocol is an elevation that permits programs to be controlled remotely without the need for them to be adapted to the distributed environment (e.g., by the introduction of special system calls to transmit network messages). In order to be more precise about enhancements and elevations, we will introduce the concept of abstract type, and then illustrate different extensions in terms of abstract types.

6.3.1 Abstract Types

The _abstract type_ is an important conceptual and practical tool for the description of complex software systems. The concept was popularized by Parnas [44, 45] and others, and is now embodied in a number of computer languages (e.g., Mesa [40], Alphard [21], and Ada [27, 28]). The discussion of COS extensibility is simplified by the use of abstract types. For completeness and because the precise meaning of "abstract type" varies in the literature, we develop the concept briefly below.

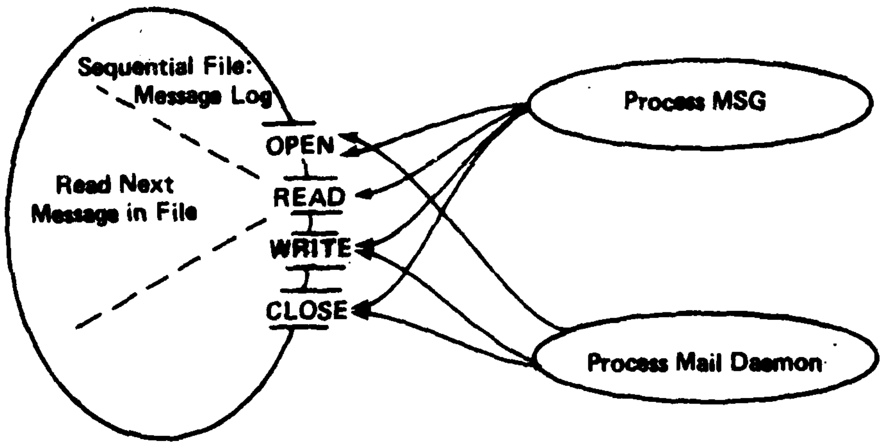


Figure 12. The Interface to Type SequentialFile

To a programmer, an abstract type appears to be a named set of _values_, together with some operations on those values. The type **SequentialFile**, for example, might have as its set of values all possible sequential data files on a particular computer system. Figure 12 lists the set of primitive operations on the abstract type **SequentialFile.**

An _object_ of a type is a name associated with a value of the type. The value of an object may change over time, as the primitive operations of the type are applied to the object. To continue the example, the value of an object MessageLog of type **SequentialFile** might change as new messages are appended to the file. All abstract types have operations to create (and destroy) objects of their type.

Abstract types are themselves objects (they can be defined formally as objects of a type **AbstractType**). An abstract type is created, may be used and modified, and eventually destroyed. An abstract type appears differently to its creators than to its users, since the former group must have access to the _representations_ of the abstract values and operations of the type. The representation of an abstract object is defined in terms of other abstract objects of different types, and their operations; of course, eventually all abstract types are derived from a set of primitive types known to the system *a priori*.

The abstract type concept is very useful for describing systems of software and hardware modules that interact only in carefully controlled ways. Because the model has been developed only recently, and because support for a uniform environment of abstract types and objects is complex, currently available operating systems do not implement them. The Hydra operating system [7] (discussed above) does implement a uniform object world, and provides a rich environment for the construction of operating system extensions.

6.3.2 Enhancements

Figure 13 shows primitive type **PhysicalFile** used in the representation of **SequentialFile**. This is an example of enhancement in the language of abstract types. Enhancements can be generated by:

1. Creating new abstract types.
2. Modifying an existing type.
3. Creating new objects of existing types.
4. By a combination of the above.

An existing type can be enhanced by the addition of new operations on its values, or the extension of previously defined operations to broader argument domains. New objects of an

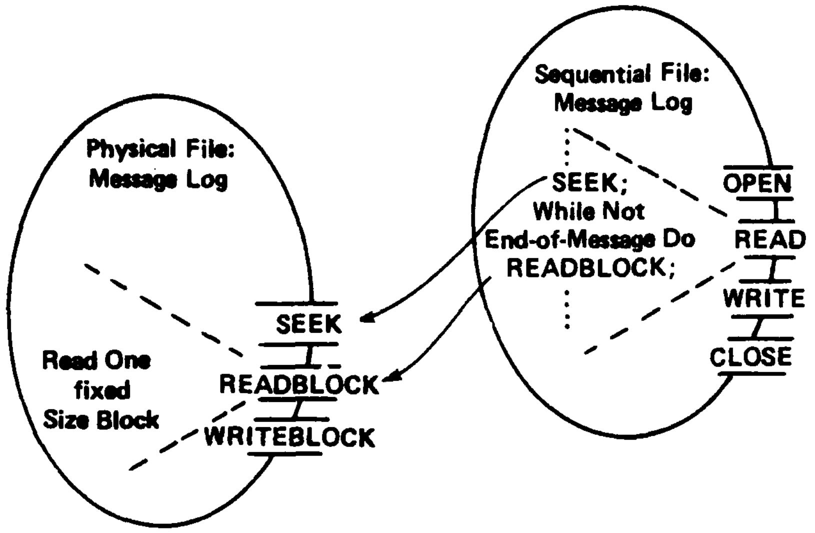


Figure 13. The Definition of SequentialFile

existing type are an enhancement insofar as clients can obtain new behavior through the operations on the object.

Even when an enhancement is implemented on an OS without full support for the abstract type model, it remains a useful thought tool for formulating extensions and establishing the relationships between modules.

6.3.3 Elevations

Ideally, the services of a COS would be structured as a collection of abstract data types. The extensibility mechanism would enable a client process P to replace a data type T used by client process Q with a type T', in such a way that Q is unaware of the substitution. Because P cannot know, in general, which operations of T are used by Q, T' must either implement or trap as an error every operation of T. (T' may define new operations not in T, if downward compatibility is not a requirement.)

There are two sizable pitfalls to this approach. First, if type T' is to be a truly transparent extension to T, it must implement a subset of the operations of T with identical semantics, as observed from Q. Demonstrating that this has been achieved is a problem in program verification and is infeasible at this time except for simple examples. Second, if the set of operations in T is large, it may be very costly to build a meaningful extension T' (i.e., one which will support many different clients like Q with few error traps).

A mechanism for elevation is extremely desirable in a COS because it enables COS local programs to be "lifted" into the global context without modification. The cost of DOS application programming may be greatly reduced if complex programming tools can be easily imported into the distributed system.

An example of elevation based on the abstract type **SequentialFile** is shown in Figure 14. Process P using the abstract type **SequentialFile** obtains only local effects from the operations Open, Read, Write,... while process Q using the elevation **SequentialFile'** may act on remote copies of a file.

A mechanism for elevation is present in the Tenex operating system in the form of JSYS traps [66]. JSYS traps allow one process to define and control the virtual machine seen by other processes. The controlling process does this by informing the Tenex monitor of its desire to "trap" certain system calls issued by the controlled processes. Whenever a controlled process issues one of the trapped calls control is transferred by the Tenex monitor to the trapping process, which may then perform

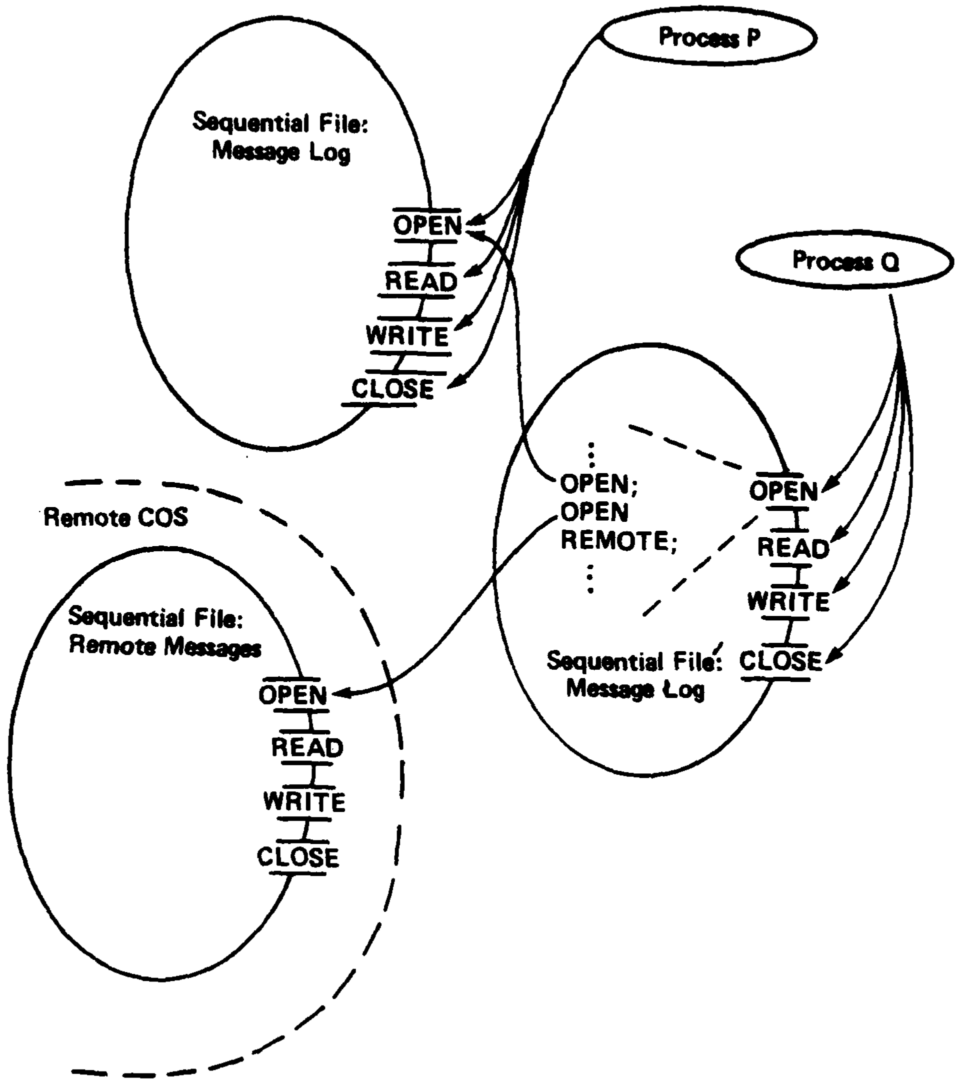


Figure 14. An Example of an Elevation

arbitrary actions including inspection of the state of the trapped process, resumption of the original system call, termination of the trapped process, and so on. The JSYS trap mechanisms was designed to support elevations for distributed computing, as part of the RSEXEC effort [65].

The Multics operating system includes the concept of dynamic linking, which can be used in a manner analogous to JSYS traps. Under dynamic linking, the binding of names to services is postponed until the first attempt by a process to use the service; at that time the name is resolved in the current context of the process. The context of the process can, under certain circumstances, be controlled by the parent of the process. By properly constructing the context of a process a parent can transparently substitute user defined services for system defined services.

The common feature of both JSYS traps and dynamic linking is that they permit a binding process (system call numbers to routines, in the case of JSYS traps, and procedure names to segment entry points, in the case of dynamic linking) to be transparently modified by a "third party". This is accomplished by some form of indirect access, although the details differ considerably. For JSYS traps, the TENEX monitor maintains a trap vector table for each process. When a trap occurs, the result may either be a transfer to the standard service routine or the awakening of the "third party" -- a user-defined process which can perform an arbitrarily complex action before restarting the trapped process. Multics employs a "linkage segment" containing indirect references to procedures; initially the references are symbolic, but after the first call the symbolic reference is replaced by a segment entry address.

The set of independently trappable service requests is fixed in the case of JSYS traps (it is just the set of system calls) whereas any symbolic procedure name can be replaced with dynamic linking. Finer control over the substitution of services is potentially provided by Multics[16]. The JSYS trap mechanism is simpler than the linkage segment mechanism in Multics (linkage segments have other important uses besides dynamic linking) and was, in fact, an enhancement to Tenex. The linkage segment concept is central to Multics and woven into the design in complex ways.

[16]Substitution of services can be thwarted by poor modularity or by permanent binding of modules by the Multics binder.

Either JSYS traps or dynamic linking can be used as a mechanism for elevation, but the ability to gain control at the proper point is only one of the obstacles to the creation of useful elevations. The completeness and correctness of the client supplied services remains a major issue, and suggests elevation is not likely to be a commonly used strategy for minor functional extensions. For major extensions, such as the construction of subsystems as discussed in the next section, it remains an important technique.

6.4 Subsystems

By a subsystem we mean an environment for user activities assembled in such a way that the user rarely has need to leave the subsystem to perform a task. The design of a subsystem is more difficult than the design of many other programs because of the requirement for completeness and coherence of functions presented to the user. Often subsystems are described entirely in one "user manual" that attempts to capture the total range of activities pursued by the subsystem user. Prominent examples of subsystems in the realm of programming systems are LISP and APL environments; many examples exist in business applications. The design and construction of subsystems is made costly by the requirement for completeness.

For example, an APL subsystem must contain much more than an APL interpreter. At a minimum, it must contain an editor for APL functions, trace and debugging facilities, and a cataloguing facility for functions. If the subsystem is to be complete in the sense that a programmer need not leave the system at all, it will probably include directory services for files, a mail system, libraries of APL functions, operations to move functions between the workspaces of different users, etc. Many of these services are probably already available to users of the APL host operating system through some other interface (e.g., the "monitor", "shell", "exec", etc.).

Subsystems can be constructed more inexpensively if the host system provides facilities for subsystem production, i.e., for the elevation of system services into subsystems. Packaging a DOS as a subsystem is a natural way to produce the DOS component processes on COS's; thus although features for subsystem construction are not directly concerned with distributed computation, they will significantly influence the cost of implementation of a DOS.

A host should have a well-defined set of conventions which can be adopted by the subsystem. Some specific areas we believe require careful consideration in a COS are conventions for:

1. **Object name syntax.** A uniform syntax for COS objects (primarily files, processes, and hosts) encourages commonality of directory, name parsing, and name validation services. A flexible convention will permit long names to be created, and should not make automatic generation of names difficult. Conventions for name patterns (so-called "wild card matches" in Tenex and TOPS-20) are valuable and should be treated in a uniform manner by COS processes.

2. **Input/Output Editing Conventions.** Operating systems support conventions for rudimentary editing of textual input and output. The input editing functions typically include backspace, cancel current line, retype current line, and perhaps a few others; the output editing functions include stop and start output and discard all output until next request for input. It is important for these conventions to be maintained across COS processes, because it is extremely irritating for users to learn different conventions for seemingly identical functions. It is less important for these conventions to be maintained across subsystems, since by assumption users do not move between subsystems very frequently.

3. **Command Scanning.** A convention for the syntax of command lines can be important. A famous example of this is the syntax of OS/360 JCL, an early use of formal syntactic specification. If the parsing services for commands are modularized and available to system clients there is increased incentive for a uniform command interface, since clients need not contain the parsing code themselves.

6.5 Design Issues

6.5.1 Transparency

Sometimes the structure of a subsystem prevents access to operations of the COS or lower level subsystems that is subsequently found to be essential at the higher level. The result is the partial redesign of the lower layers to give the subsystem client access to the needed operations. The redesign can be costly and difficult because it spans multiple levels of the system.

A principle intended to eliminate this problem is espoused in [46]. The principle requires that higher levels of software be transparent, in the sense that a layer will not prevent a higher layer from achieving any valid state in the layers below it. The principle of transparency is in direct contradiction with the principle of information hiding. The compromise reached between them is an issue for serious consideration in the design of a COS.

As an example of where the problem occurs, consider the COS interface to the communication network. Should the COS permit clients (all clients? authorized clients?) to directly control the network interface? A DOS may need low level control over the protocols transmitted, or error correction techniques to be employed. Low level access, on the other hand, often implies the ability to disrupt other clients' use of the interface. For this reason it should only be permitted if the DOS subsystem is trusted by the COS.

6.5.2 Economics

COS mechanisms have associated costs. It is difficult or impossible to separate the costs of features specifically introduced to facilitate participation in a distributed system from those necessary or desirable in a centralized OS.

For instance, a mechanism for elevation is desirable in a COS. What is the cost of the JSYS trap mechanism, or of the dynamic linking facility in Multics? The JSYS trap mechanism was introduced to support elevations for a distributed application, but how much of its cost should be considered specifically as distributed system support, when it can also be used for single host systems?

The cost of COS mechanisms can generally be divided according to the time at which the cost is incurred. There is a component of the cost incurred when the mechanism is designed and implemented; another when a process is constructed using the mechanism (e.g., for static data structures); and some cost each time the mechanism is used. There should also be an awareness that processes on the COS which do not use the mechanism may suffer some overhead (e.g., for indirection through the linkage segment in Multics). The tradeoffs among the various cost alternatives can only be discussed in a specific design context.

A factor which influences the cost of an elevation mechanism is the precision of specification of the trapping condition. If the trapping condition can only be stated coarsely (e.g., "trap all file OPEN system calls") then some overhead is accumulated when the trap occurs unnecessarily and execution must be resumed as if the elevation were not present. If the trapping condition can be specified precisely (e.g., "trap any OPEN on file MessageLog in directory <wmacgregor>") the overhead can possibly be reduced. In this regard we would expect the dynamic linking mechanism to be a more powerful mechanism with lower overhead than JSYS traps, because it allows the trapping condition to be specified in the domain of the client program (i.e., an arbitrary procedure name).

6.6 Recommendations

If the construction of a COS from the beginning is considered:

1. A general mechanism for producing elevations should be part of the COS design
2. Control of COS resources (especially primary memory and processor time) should be possible from client processes.

The Hydra operating system is an important model (the only one we know of combining both aspects) for achieving these properties.

If an existing OS is to be adapted to become a COS, the selected OS should be chosen for flexibility in the areas of:

1. Process structuring and interprocess communication.
2. Directory services for OS objects.
3. Control by client processes over system resources.
4. Features for subsystem construction.

7. PLAN FOR PHASE II

Our plan for Phase II of the DOS Design study is to develop a prototype DOS design and to perform further development of global resource management strategies.

7.1 Characterization of task

- To develop a prototype design of both a DOS and the underlying support of the COS: O.S. primitives, resources, servers, ..., primarily in the area of reliability and resource management. Due to the limited remaining resources of the contract, limits will exist on the depth to which the topics may be explored.
- To develop mechanisms for supporting various global resource management policies. For selected mechanisms a quantitative study of the benefits of global resource management will be performed using simulation and mathematical modeling techniques. The models will concentrate on physical resources, these being processors, memory, and the shared communication subsystem.
- To do this work from the perspective of one specific application based on the Field Control Unit example which has requirements and constraints very similar to those of the automation of office procedures, and to some projected data processing components of the Tactical Air Force. Limiting ourselves to a particular application will help bound the level of effort needed.

7.2 Reason for performing task

A conclusion from Phase I of this study is that many different types of reliability mechanisms have been proposed, but very few operating systems have integrated such mechanisms into their set of general purpose primitives. Phase I also indicated that almost no relevant work has been done towards the aim of managing distributed resources to achieve global objectives.

Phase I surveyed known reliability techniques, but did not elaborate the ways they might be used. Phase I also identified a number of factors important to global resource management, but did not develop specific strategies. Meaningful evaluations of the effectiveness of reliability mechanisms and global resource management are strongly dependent on the application and the architecture on which the application is implemented.

We feel that the critical problems in these two subject areas involve the integration of the mechanisms into a useful whole. Therefore, Phase II will be aimed at selecting and integrating into a prototype design: 1) an appropriate application; 2) a distributed architecture as the implementation base; and 3) a suitable selection of reliability and global resource management mechanisms.

7.3 Topics to be investigated

- o Description of the target application.
- o Description of the distributed architecture implementation base.

- Selection and rationale for a set of reliability and global resource management mechanisms based on the application and implementation base.

- In the area of global resource management, special emphasis on:

  - Definitions of _priority_, _class of service_, and _timeliness_ as they apply to the formulation of global resource management policies.

  - The precise formulation of a small, representative set of global resource management strategies.

  - The definition of expected system workloads based on the characterization of the application and its probable usage patterns.

  - The investigation of the performance properties of strategies in the representative set using simulation and mathematical modeling techniques.

- Description of prototype COS and DOS mechanisms in our two areas of specialized interest and other areas of O.S. functionality.

- Description of the manner in which these integrated facilities can be used to implement the target application.

7.4 Expected Benefits and Results

- Evidence that the selected reliability and resource management techniques are compatible and appropriate in an integrated design.
- Exploration of the implementation problems that would result in using such mechanisms in an integrated O.S.
- Explicit ways in which reliability mechanisms such as 1) atomic actions, 2) flexible bindings, and 3) distributed coordination are used in a distributed application.
- A catalogue of global resource management strategies containing relative performance measures.

REFERENCES

[1] T. Anderson, P. A. Lee, S. K. Shrivastava.
System Fault Tolerance.
Technical Report, Univ. of Newcastle upon Tyne (UK), July, 1978.

[2] J. F. Bartlett.
A 'Non-Stop' Operating System.
In Proc. Eleventh Hawaii International Conference on System Sciences, pages 103-117. University of Hawaii, Honolulu, Hawaii, December, 1978.

[3] A. J. Bernstein & J. C. Sharp.
A policy-driven scheduler for a time-sharing system.
Communications of the ACM 14(2):74-78, February, 1971.

[4] L. A. Bjork.
Recovery Scenario for a DB/DC System.
In Proc. ACM National Conference, pages 142-146. ACM, 1973.

[5] P. Brinch Hansen.
The Nucleus of a Multiprogramming System.
Communications of the ACM 13(5):238-250, April, 1970.

[6] P. BrinchHansen.
The Programming Language Concurrent Pascal.
IEEE Trans. on Software Engineering SE-1(2):199-207, June, 1975.

[7] E. Cohen & D. Jefferson.
Protection in the Hydra operating system.
Operating Systems Review 9(5):141-160, November, 1975.
Proceedings of the Fifth Symposium on Operating Systems Principles.

[8] S. D. Crocker.
The National Software Works: A New Mechod for Providing Software Development Tools Using the ARPANET.
In Proc. Meeting on 20 Years of Computer Sacience. Consiglio Nazionale delle Richerche Instituto Di Elaborazione Della Informazione, June, 1975.

[9] C. T. Davis.
Recovery Semantics for a DB/DC System.
In Proc. ACM National Conference, pages 136-141. ACM, 1973.

[10] E. W. Dijkstra.
Cooperating Sequential Processes.
Technical Report, Technological University, Eindhoven, The Netherlands, 1965.
Reprinted in Programming Languages, F. Genuys, ed., Academic Press, New York, N.Y., 1968.

[11] C. A. Ellis, G. J. Nutt.
Office Information Systems and Computer Science.
Computing Surveys 12(1):27-60, March, 1980.

[12] L. D. Erman, F. Hayes-Roth, V. R. Lesser, & D. R. Reddy.
The Hearsay-II speech-understanding system: integrating knowledge to resolve uncertainty.
ACM Computing Surveys 12(2):213-253, June, 1980.

[13] D. J. Farber, et. al.
The Distributed Computing System.
In Proc. IEEE Computer Society Internaltional Conference, COMPCON-73, pages 31-34. IEEE, 1973.

[14] D. J. Farber.
A Ring Network.
Datamation 21(2):44-46, February, 1975.

[15] J. Feldman.
High level programming for distributed computing.
Communications of the ACM 22(6):353-368, June, 1979.

[16] A. K. Fitzgerald, B. F. Goodrich.
Data Management for the Distributed Processing Programming Executive (DPPX).
IBM Systems Journal 18(4):547-564,1979.

[17] R. W. Floyd.
Nondeterministic Algorithms.
Journal of the ACM 14(4):636-644, October, 1967.

[18] J. B. Goodenough.
Exception Handling: Issues and a Proposed Notation.
Communications of the ACM 18(12):683-696, December, 1975.

[19] N. C. Goodwin, J. Mitchell, & P. S. Tasker.
Concept of operations for message handling at CINCPAC.
Technical Report MTR-3323, Mitre Corporation, October, 1976.

[20] F. E. Heart, R. E. Kahn, S. M. Ornstein, W. R. Crowther, D. C. Walden.
The Interface Message Processor for the ARPA Computer Network.
Proc. SJCC 36:551-567,1970.

[21] P. Hilfinger, G. Feldman, R. Fitzgerald, I. Kimura, R. L. London, KVS Prasad, VR Prasad, J. Rosenberg, M. Shaw, & W. A. Wulf.
An informal definition of Alphard.
Technical Report CMU-CS-78-105, Carnegie Mellon University, February, 1978.

[22] C. A. R. Hoare.
Monitors: An Operating SYstem Structuring Concept.
Comm. of the ACM 17(10):549-558, October, 1974.

[23] A. L. Hopkins, Jr., T. B. Smith, III, & J. H. Lala.

FTMP--a highly reliable fault-tolerant multiprocessor for aircraft.
Proceedings of the IEEE 66(10):1221-1239, October, 1978.

[24] J. J. Horning.
A Taxonomy of Error Responses.
Technical Report SRM/41, University of Newcastle Upon Tyne, May, 1973.

[25] J. G. Hunt.
Messages in typed languages.
SIGPLAN Notices 14(1):27-45, January, 1979.

[26] IBM Corporation.
IBM System/360 PL/I Reference Manual.
Technical Report C28-8201-0, IBM Corporation, 1967.

[27] J. Ichbiah, et. al.
Preliminary Ada Reference Manual.
ACM SIGPLAN Notices 14(6):entire issue, June, 1979.
Part A.

[28] J. D. Ichbiah, J. G. P. Barnes, J. C. Heliard, B. Krieg-Brueckner, O. Roubine, B. A. Wichmann.
Rationale for the design of the Ada programming language.
SIGPLAN Notices 14(6):entire issue, June, 1979.
Part B.

[29] J. A. Katzman.
A Fault-Tolerant Computing System.
In Proc. Eleventh Hawaii International Conference on System Sciences, pages 85-102. University of Hawaii, Honolulu, Hawaii, December, 1978.

[30] S. C. Kiely.
An Operating System for Distributed Processing -- DPPX.
IBM Systems Journal 18(4):507-525,1979.

[31] L. Lamport.
Time, clocks, and the ordering of events in a distributed system.
Communications of the ACM 21(7):558-565, July, 1978.

[32] B. W. Lampson, & Howard E. Sturgis.
Crash recovery in a distributed data storage system.
Technical Report, Xerox PARC, April, 1979.

[33] B. W. Lampson, R. F. Sproull.
An Open Operating System for a Single-User Machine.
In Proc. of the Seventh Symposium on Operating Systems Principles, pages 98-105. ACM, 1979.

[34] R. Levin, E. Cohen, W. Corwin, F. Pollack, & W. Wulf.
Policy/mechanism separation in Hydra.
Operating Systems Review 9(5):132-140, November, 1975.
Proceedings of the Fifth Symposium on Operating Systems Principles.

[35] B. G. Lindsay, P. G. Selinger, C. Galtieri, J. N. Gray, R. A. Lorie, T. G. Price, F. Putzolu, I. L. Traiger, B. W. Wade.
Notes on Distributed Databases.
Technical Report RJ2571, IBM Research Laboratory, San Jose, CA, July, 1979.

[36] D. B. Lomet.
Process Structuring, Synchronization, and Recovery Using Atomic Actions.
ACM SIGPLAN Notices 12(3):128-137, March, 1977.

[37] R. M. Metcalfe & D. R. Boggs.
ETHERNET: distributed packet switching for local computer networks.
Communications of the ACM 19(7):395-404, July, 1976.

[38] R. Millstein, R. Shapiro.
NSW Reliability Plan.
Technical Report ???, Massachusetts Computer Associates, ???, ???.

[39] G. Milne & R. Milner.
Concurrent processes and thier syntax.
Journal of the ACM 26(2):302-321, April, 1979.

[40] J. G. Mitchell, W. Maybury, & R. Sweet.
Mesa language manual.
Technical Report CSL-79-3, Xerox PARC, 1979.
Version 5.0.

[41] W. A. Montgomery.
Polyvalues: A Tool for Implementing Atomic Updates.
In Proc. of the Seventh Symposium on Operating Systems Principles, pages 143-149. ACM-SIGOPS, Pacific Grove, CA., December, 1979.

[42] T. H. Myer & D. W. Dodds.
Notes on the development of message technology.
In Proceedings of the Berkeley Workshop on Distributed Data Management and Computer Networks, pages 144-154. Lawrence Berkeley Laboaratory and United States Energy Research and Development Administration, May, 1976.

[43] C. D. Pack.
Configuring a private line circuit according to the hi-lo tariff - a minimal Steiner tree problem.
In Proceedings of the Fourth Data Communications Symposium, pages 6-12 to 6-18. Association for Computing Machinery, October, 1975.

[44] D. L. Parnas.
On the criteria to be used in decomposing systems into modules.
Communications of the ACM 15(12):1053-1058, December, 1972.

[45] D. L. Parnas.
A technique for software module specification with examples.
Communications of the ACM 15(5):330-336, May, 1972.

[46] D. L. Parnas & D. P. Siewiorek.
Use of the concept of transparency in the design of hierarchically structured systems.
Communications of the ACM 18(7):401-408, July, 1975.

[47] B. Randell.
System Structure for Software Fault Tolerance.
IEEE Transactions on Software Engineering 1(2):220-232, June, 1975.

[48] B. Randell, P. A. Lee, P. D. Treleaven.
Reliability Issues in Computing System Design.
Computing Surveys 10(2):123-165, June, 1978.

[49] E. G. Rawson.
Application of fiber optics to local networks.
In N. B. Meisner & R. Rosenthal (editors), *Proceedings of the Local Area Communications Network Symposium*, pages 155-168. Mitre Corporation and the National Bureau of Standards, May, 1979.

[50] D. P. Reed.
Implementing Atomic Actions on Decentralized Data.
In *Proc. of the Seventh Symposium on Operating Systems Principles*, pages 163. ACM-SIGOPS, Pacific Grove, CA., December, 1979.

[51] D. A. Rennels.
Distributed fault-tolerant computer systems.
Computer 13(3):55-65, March, 1980.

[52] L. G. Roberts, B. D. Wessler.
Computer Network Development to Achieve Resource Sharing.
In *Conference Proceedings, Vol. 36*, pages 543-549. AFIPS, 1970 SJCC.

[53] B. Rothfarb & M. Goldstein.
The one-terminal Telpak problem.
Operations Research 19:156-169, January-February, 1971.

[54] A. Rybczinski, B. Wessler, R. Despres, & J. Wedlake.
A new communication protocol for accessing data networks--the international packet mode interface.
In *Proceedings of the National Telecommunications Conference 45*, pages 477-482. AFIPS, June, 1971.

[55] J. H. Saltzer & K. T. Pogran.
A star-shaped ring network with high maintainability.
In N. B. Meisner & R. Rosenthal (editors), *Proceedings of the Local Area Communications Network Symposium*, pages 179-189. Mitre Corporation and the National Bureau of Standards, May, 1979.

[56] J. H. Saltzer.
Comments on Distributed Systems.
In talk given at 7th Symposium on Operati[illegible] Systems Principles, pages . ACM-SIGOPS, 1979.

[57] D. G. Severance, G. M. Lohman.
Differential Files: Their Application to the Maintenance of Large Databases.
ACM Transactions on Database Systems 1(3):256-267, September, 1976.

[58] S. K. Shrivastava, J-P. Banatre.
Reliable Resource Allocation Between Unreliable Processes.
IEEE Trans. on Software Engineering SE-4(3):230-241, May, 1978.

[59] S. K. Shrivastava.
Concurrent Pascal with Backward Error Recovery: Language Features and Examples.
Software -- Practice and Experience 9(12):1001-1020, December, 1979.

[60] J. Stern.
Automatic File Backup in a Computer Utility.
Technical Report MAC-TR-117, MIT Laboratory for Computer Science, September, 1973.

[61] M. Stonebreaker, E. Wong, & P. Kreps.
The design and implementation of INGRES.
Transactions on Database Systems 1(3):, September, 1976.

[62] R. Stotz, R. Tugender, D. Wilczynski, & D. Oestreicher.
SIGMA--an interactive message service for the Military Message Experiment.
In Conference Proceedings, Vol. 48, pages 839-846. AFIPS, 1979 NCC.

[63] A. Takagi.
Concurrent and Reliable Updates of Distributed Databases.
Technical Report MIT/LCS/TM-144, MIT, Nobember, 1979.

[64] R. H. Thomas.
A Solution to the Concurrency Control Problem for Multiple Copy Data Bases.
In Proc. 1978 Spring COMPCON, San Francisco, pages -. COMPCON, February, 1978.

[65] R. H. Thomas.
A resource sharing executive for the ARPANET.
In Conference Proceedings, Vol. 42, pages 155-163. AFIPS, 1973 NCC.

[66] R. H. Thomas.
JSYS traps--a TENEX mechanism for encapsulation of user processes.
In Conference Proceedings, Vol. 44, pages 351-360. AFIPS, 1975 NCC.

[67] W. N. Toy.
Fault-Tolerant Design of Local ESS Processors.
In *Proc. of the IEEE*, pages 1126-1145. IEEE, 1978.

[68] T. Turton.
The management of operating system state data.
Operating Systems Review 14(2):21-24, April, 1980.
Reprinted from *Quaestiones Informaticae*, V 1 N 4, September 1979, a publication of the Computer Society of South Africa.

[69] J. M. Verhofstad.
Recovery Techniques for Database Systems.
Computing Surveys 10(2):167-195, June, 1978.

[70] F. C. H. Waters.
Design of the IBM 8100 Data Base and Transaction Management Systems -- DTMS.
IBM Systems Journal 18(4):565-581,1979.

[71] J. H. Wensley, L. Lamport, J. Golberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, & C. B. Weinstock.
SIFT: the design and analysis of a fault-tolerant computer for aircraft control.
Proceedings of the IEEE 66(10):1240-1255, October, 1978.

[72] M. V. Wilkes.
The Cambridge digital communication ring.
In *Proceedings of the Local Area Communications Network Symposium*. Mitre Corporation, Boston, May, 1979.

[73] W. A. Wulf & S. P. Harbison.
Reflections in a pool of processors--an experience report on C.mmp/Hydra.
In *Conference Proceedings, Vol. 47*. AFIPS, 1978.

[74] W. Wulf, R. Levin, & C. Pierson.
Overview of the Hydra operating system development.
Operating Systems Review 9(5):122-131, November, 1975.
Proceedings of the Fifth Symposium on Operating Systems Principles.

ILMED

8